TEXT STEGANOGRAPHY

BY

ADAM DOUGLAS CAIN

B.S.E., University of Iowa, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

APRIL 1996

WE HEREBY RECOMMEND THAT THE THESIS BY

ADAM DOUGLAS CAIN

ENTITLED TEXT STEGANOGRAPHY

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

O-517

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GRADUATE COLLEGE DEPARTMENTAL FORMAT APPROVAL

THIS IS TO CERTIFY THAT THE FORMAT AND QUALITY OF PRESENTATION OF THE THESIS SUBMITTED BY ADAM DOUGLAS CAIN AS ONE OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE ARE ACCEPTABLE TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING.

APRIL 25, 1996
Date of Approval

Julian I Bach
Departmental Representative

ABSTRACT

Since the exchange of encrypted data in interpersonal electronic messages is both rare and easily detected, steganographic techniques are needed to ensure that private communications do not raise suspicion. Previous work in this area is largely inapplicable to environments such as the Internet, as these schemes require the exchange of large data sets in the form of image or sound files, or the sharing of large databases used for the steganographic encoding and decoding. A class of novel systems is presented here which aims to provide a practical solution for steganography over text-based channels with minimal shared information required. The implementation of one of the four presented systems is described and initial experimental results are reported.

ACKNOWLEDGMENTS

The author would like to profusely thank Dr. Ricardo Uribe for his consistent support and interest throughout the duration of this project. This work has also benefited greatly from the ideas and suggestions of Rick Hoselton and Robert McGrath. Finally, it is likely that the project would not have reached this point without the enthusiasm and interest shared by friends, especially William Gillespie and Joseph Futrelle.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 Motivation

Today most communication occurs over insecure channels. The information exchanged by phone, postal mail, electronic mail or similar means may be subject to any number of forms of eavesdropping or monitoring. This observation is based less on paranoia than on plain common sense.[1]

The threats and risks involved in communicating in the paper world or via the telephone network are fairly well-understood, and they are limited in large part by the physical effort required by an eavesdropper wishing to monitor particular exchanges. However, in the realm of electronic communications across "open" networks such as the Internet, the threats are far greater. It is surprisingly easy to monitor or even intercept digital message exchanges on a public, packet-switched network since the communication paths employed are neither controlled nor secure. And considering that eavesdroppers not only have access to these communication links, but they may also have substantial computing resources, the threat to personal privacy posed by automated monitoring tools becomes a serious issue.

The rich field of cryptography is concerned with how information can be scrambled in such a way that only the intended recipient(s) can descramble and make use of it. Fortunately, the decades of intense study devoted to this problem have led to a number of solutions which are thought to be adequate. There are a number of seemingly unbreakable algorithms for encrypting and decrypting digital data. These are even available to the general public in the form of freely available software, e.g., the "Pretty Good Privacy" package by P. Zimmerman [2].

However, there is an important problem with the use of cryptography in personal communications: it is easily detected. Encrypted data are usually quite identifiable by its statistical characteristics alone. If enciphered communications employ, as is often the case, tell-tale headers such as "BEGIN PRIVACY-ENHANCED MESSAGE," the situation is all the more obvious. And since encrypted communications are currently the exception rather than the rule, its use may be perceived as conspicuous. Even if the message is unbreakably encrypted, the aroused suspicion may lead resourceful attackers to pursue alternative means to violating the privacy of the communicating parties.

---

[1] The two are obviously interrelated, as observed by Allen [1].

How can two parties communicate in a way that is both secure and inconspicuous? A related question that has extreme political relevance currently is "Can two parties use cryptography without its use being easily detected?" In the recent debate about making illegal the use of certain encryption algorithms (such as those which are unbreakable by federal agencies), the feasibility of detecting the use of forbidden encryption algorithms is tacitly assumed. After all, it is a practical impossibility to regulate something that is indetectable.

Answering these questions requires us to investigate the strange and interesting field of steganography.

1.2 Steganography

"Steganography" (from Greek, meaning hidden writing) is the art of sending messages in such a way that observers other than the intended recipient(s) do not know that a private communication is transmitted. The hidden messages themselves may or may not be encrypted. The important characteristic of steganographic systems is that the actual messages exchanged look innocuous to the eavesdropping observer. Combining cryptography with steganography ensures that the plaintext of the hidden message is not accessible to the attacker, even if she suspects that such a message exists by observing the communication.

1.2.1 Historical examples

History provides us with many examples of steganography, dating back to the time of the Ancient Greeks. The Histories of Herodotus describe how a secret message was sometimes tattooed on the shaven head of a messenger [3], [4]. Once the hair had grown back, the messenger was dispatched. Upon reaching his destination, the messenger's head was again shaved to reveal the hidden message. While this method allowed secret messages to be carried past enemy sentries without raising suspicion, it can be said to suffer from a relatively low bandwidth.

More modern examples of steganographic systems draw largely from secret communications in the contexts of warfare or espionage. A variety of "invisible ink" schemes have been used to conceal messages within innocent-looking documents. During World War II, the Nazis developed a "microdot" technology that allowed photographs the size of a printed period to be enlarged to produce images with the clarity of a standard-sized typewritten page [3], [4]. Many other examples can be found in [3].

1.2.2 Modern steganography

Somewhat surprisingly, the academic treatment of steganography has been rather sparse. Some theoretical work has been done in the closely related area of subliminal channels [5], and the technique of using spread-spectrum technology for indetectable wireless communication can be considered one form of steganography. However, unlike the case of cryptography, there are few theoretical results which help us produce practical tools for wide-scale steganography in the context of modern communication networks. Of course it is also possible that such theoretical results have been produced and they were simply too effective at hiding their own messages.

Figure 1.1 depicts the communication model of interest for contemporary steganography. Note that we are assuming that the steganographic system is used to exchange encrypted data. This is based on the assumption that if the steganographic scheme fails, the communicating parties will still want the messages to be unreadable by the attacker. The cryptographic scheme may be either symmetric (the same key used for encryption and decryption) or asymmetric (a message is encrypted using the recipient's public key). This choice is usually independent of the steganographic system.

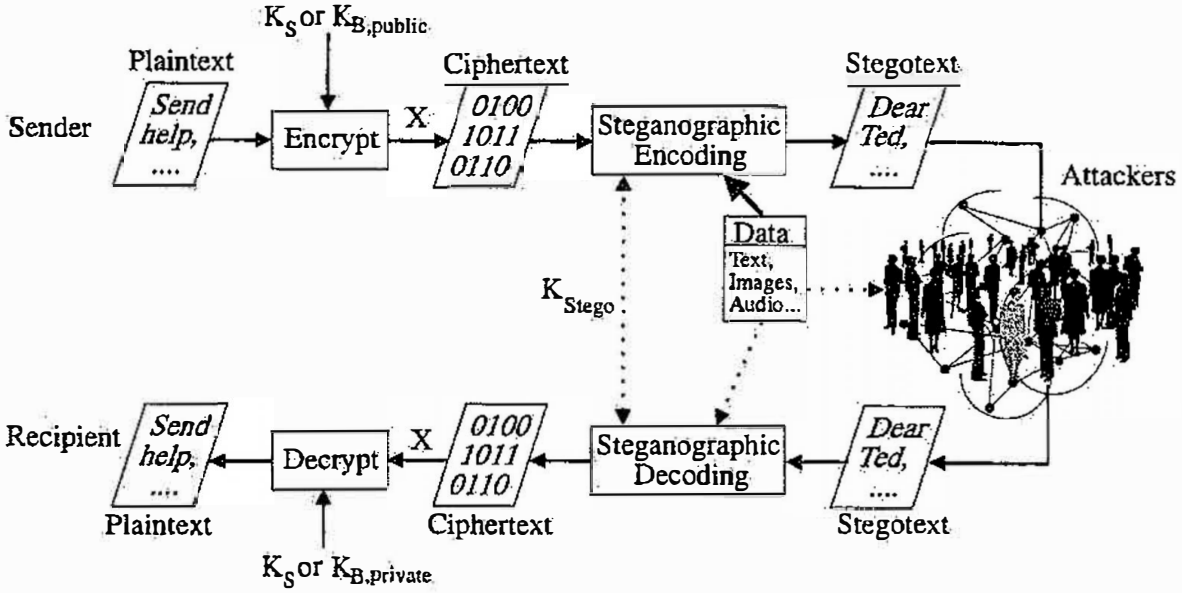

Figure 1.1. Communication model of modern steganography

The "stegotext" may be text, image data, sound data, or data of any number of other formats. The schemes introduced in this document hide messages in text. And as the class of these novel systems has been given the name "bonzo-coding,"[2] we will use the word "bonzotext" to refer to the stegotext that is exchanged by parties using these particular schemes.

The attacker is assumed to have access to the following information:

- All stegotext exchanged
- Full knowledge of the algorithms used for both cryptography and steganography
- Any desired statistical information about the data format used as stegotext.

The requirement that the attacker know everything about the algorithms employed is called "Kerckhoff's assumption [6]," and is standard in cryptographic disciplines. For a practical steganographic system, we must not depend on the secrecy of the choice of information hiding algorithms.

A steganographic scheme may or may not require keying material in addition to that which is used by the cryptosystem. Ideally, keys would not be needed for encoding or decoding the stegotext, and this (along with the Kerckhoff assumption) implies that the attacker would have direct access to the ciphertext. We will see that this does not always compromise the security of the steganographic system. In general, the use of keys with steganography can be helpful in preventing the attacker from obtaining information used to detect hidden messages in the stegotext.

It is also worth noting that while the attacker is assumed to possess an extensive corpora and accurate statistics of the data type used for hiding messages, he does not necessarily have the same data held by the sender and/or receiver. For example, the attacker may have a huge archive of digital audio samples, along with high-quality statistics and models describing them; however, the sender and perhaps also the receiver may use particular audio samples that the attacker does not possess.

Finally, note that the statistical properties of the ciphertext X are assumed to be known by the attacker, and we may reasonably assume that these statistics match those of white noise. That is, any decent encryption algorithm produces data which may be accurately modeled as a random sequence with each bit or byte being independent and uniformly distributed. The independence of each bit means that guessing a particular bit value of X is not aided by knowing any of the other bit values.

[2] By naming these steganographic systems after the primate of movie fame, the author hopes to emphasize their architectural simplicity.

1.2.3 Objectives

To study the effectiveness and practicality of different steganographic systems, we use the following set of objectives, listed in approximate order of importance:

1. Unambiguous encoding

   The recipient must be able to decode the stegotext without error and recover the ciphertext (or plaintext, when encryption is not used separately) intact.

2. Minimal shared information

   It would be impractical to require that communicating parties share large amounts of secret data in order to communicate innocuously. It might be acceptable to require that they share a common passphrase. However, the sender and receiver must not be made to share large databases (e.g., sets of images or long wordlists needed for the steganographic system), particularly if discovery of these files would incriminate the communicating parties, or if they can only be used for a small number of steganographic exchanges.

3. Indetectability

   Simply put, attackers must not be able to easily tell the difference between stegotext and communications without hidden messages. It is of primary importance that the attacker not be able to use automatic techniques, such as computerized statistical analyses, to reliably detect when the communication involves steganography.

4. Maximal bandwidth

   The amount of stegotext required to send one byte of ciphertext should be minimized. Acceptable expansion ratios will depend upon the nature of the channel (whether it is based on text, digital audio, image data, etc.).

5. Time efficiency

   The time required to perform the steganographic encoding and decoding should also be minimized. This includes both the computation time required to encode or decode a particular message, as well as the human effort involved in configuring the system to exchange one or several messages.

As we survey steganographic systems, we will notice several tradeoffs inherent in their design. Usually, the systems with the least shared information and highest time efficiency are also the most detectable. Similarly, to have a maximally indetectable coding scheme, we frequently require a great deal of shared information, computation time or preparation effort. A practical steganographic system must achieve a balance of these parameters appropriate for the communication environment of interest.

1.2.4 Steganography with image and sound data

Recently, there has been a fair amount of attention devoted to the problem of how to embed encrypted data within digital images or audio data. A number of systems that accomplish this are available today [4], and most are based on the premise that the format used to store image and audio data is usually of such high precision that slight alterations of the data itself will not be perceivable to the human observer.

The typical scenario for audio steganography is as follows (the case of image data is similar). The sender generates or chooses a digital audio file of sufficient length and resolution. Using a steganographic encoder, the least significant bit(s) of all or some of the audio samples are replaced with the bits of the ciphertext and the resulting audio file is sent to the recipient. To the human listener, this new audio file will be practically indistinguishable from the original version. In order to decode the data, the recipient must either have the original audio file (in which case, the bits of the ciphertext are found by comparing the received with the original data), or know which audio samples contain bits of the ciphertext. Note that the data format must be such that the ciphertext is recovered without error, which usually requires the use of formats that are either uncompressed or use lossless compression.

There are currently several available software packages for hiding ciphertext in images and sound using common data formats, and most are surveyed in [4]. The more advanced systems employ a random sequence generator (seeded with a passphrase known to sender and receiver) to select the locations of samples whose least-significant bit(s) are replaced by bits of the ciphertext. All of these systems are based on the questionable assumption that the statistics of the least-significant bits in audio or image data are indetectably similar to those of ciphertext. A thorough statistical analysis of the nature of image or audio files before and after these steganographic encodings has yet to be published.

However, the most problematic aspect of these systems, with respect to their use in private communications between parties on the Internet, is that they obviously require the exchange of (potentially large) image or sound files. Currently, this sort of multimedia-media communication is still somewhat rare, and many of us have *never* before sent an audio or image file to someone else. For such an individual to suddenly start exchanging a large amount of graphics or digital audio might be significantly conspicuous. Furthermore, we must use images or audio files for which the attacker does not have the original versions; otherwise, detection would be as simple as finding the original file and comparing it with the transmitted version. Thus, we effectively require that each sending party generate large collections of image or audio data and keep these collections secret from attackers (i.e., we cannot simply use selections from online graphics archives).

In fact, the vast majority of electronic, person-to-person communications in environments such as the Internet consists entirely of textual information. So while image-based and audio-based steganographic tools are very useful, they do not offer a solution that is immediately deployable and usable on a wide scale in the current network environment.

An interesting variation on the image-based scheme involves hiding information by means of varying the spacing of words on a document in PostScript[3] or a similar page-layout format [7]. This work has received a fair amount of study and is of prime interest to publishers wishing to embed indetectable "watermarks" in documents. Unfortunately, documents of these types are also not regularly exchanged as a part of interpersonal communications.

To allow for practical steganography in environments such as today's Internet, we must turn to techniques which hide information in text. For the remainder of this document, we will concern ourselves only with text-based approaches.

1.3 Previous Work

1.3.1 Simple text steganographic schemes

It is useful to consider several interesting, *ad-hoc* text steganographic systems that can either be found in literary works, or contrived using simple, computer-assisted methods. These systems are not themselves useful for wide-scale, indetectably secure communications; however, they do illustrate the range of text manipulation techniques available to us for these purposes.

[3] PostScript is a trademark of Adobe Systems, Inc.

1.3.1.1 Acrostics and subliminal sentences

A number of writing styles involve the hiding of messages within text, with varying levels of detectability. If we allow these messages to include higher-order concepts such as the meaning of a particular story (i.e., the stegotext being the same as the "subtext"), there are innumerable examples of this in poetry and literature in general. For our purposes, we restrict our attention to steganographic schemes that embed specific data.

An "acrostic" is a form of writing (usually poetry) in which the letters of the hidden message are found in the concatenation of the first letter of the first word in every line. Lewis Carroll, author of Alice in Wonderland and mathematician as well, was particularly fond of this style and its variations. One of his acrostic poems is shown in Fig. 1.2. [8]

Love Among The Roses

"Seek ye Love, ye fairy-sprites?
  Ask where reddest roses grow.
Rosy fancies he invites
And in roses he delights,
  Have ye found him?" "No!"

"Seek again, and find the boy
  In Childhood's heart, so pure and clear."
Now the fairies leap for joy,
  Crying, "Love is here!"

"Love has found his proper nest;
  And we guard him while he dozes
In a dream of peace and rest
  Rosier than roses."

Jan. 3, 1878

Figure 1.2. An acrostic by Lewis Carroll

Taking the first letter of each line, we learn the name of the inspiration of this poem, one Sarah Sinclair. Some of Carroll's other poems used different letters of each line and a few of these works require nontrivial effort to discover the encoding algorithm.

"Subliminal sentences" carry this idea a step further by encoding a message using the first letter of every word. In the news poem shown in Fig. 1.3, concatenating the first letter of each word gives rise to a new text.

> Individuals distill essentialized nativist types into themselves, yes. People organize limited identities, toppling into competing sisterhoods. Protect equal rights, pre-empt established types, universally attend to everyone. "Freedom" establishes the isolated, select, haphazard individual's zealously erratic diagonal. Individual nihilist justice: United States totalitarianism internalizes competitive economics socially.

Figure 1.3. A subliminal sentence news poem by William Gillespie

These particular writing styles are obviously not well-suited for private communications, as their plaintext is easily discovered. However, note that the message in Fig. 1.4 was actually sent by a German spy during World War II [3], [4]:

> Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by-products, ejecting suets and vegetable oils.

Figure 1.4. Variation on subliminal sentence sent during WWII

Taking the *second* letter in each word produces the message, "Pershing sails from NY June I," where the final "i" is interpreted as a "1."

We could consider a scheme in which acrostics or subliminal sentences are used to encode encrypted data, rather than plaintext. For example, four-bit chunks of the ciphertext could be mapped to 16 of the most frequently used first letters of English words. The resulting expansion ratio for the subliminal sentence approach would be approximately 10:1 (assuming an average of five characters per word), which is not extreme. It is even conceivable that a computer could generate natural language with these constraints, which would reduce the human effort required for each message. The main concern is that the subliminal sentences *look* like human-written text, both statistically and qualitatively. The bonzo-coding schemes introduced in this document are, in some ways, generalizations of this approach which allow us to address this concern.

1.3.1.2 Space-coding

A trivially simple method of encoding data in human-generated text files is to add a variable number of spaces at the end of each line. In this way, a given line of text may have between one and seven spaces appended, depending on the value of each 3-bit chunk of the ciphertext. To the casual eavesdropper, the stegotext will look ostensibly the same as the original text, when displayed using most text editors.

This is an example of a system that is completely detectable if the steganographic encoding and decoding algorithms are known to the attacker. The fact that most text messages exchanged in email and the like do not contain spaces at the end of lines would make the stegotext quite conspicuous.

1.3.1.3 Smooch-coding

In the domain of human-generated text, wherever there are multiple alternatives that are considered equally valid, there is potential for information hiding. To illustrate this useful heuristic, we consider the case of multiple spellings of a word. Anyone who has witnessed or participated in the exchange of flowery messages between two sweethearts has probably noticed that there is no rigid standard for the spelling of certain onomatopoetic outbursts. If the choice of spelling for each instance of these words is a function of the value of the ciphertext, then we have an adorable steganographic system.

The author has implemented a crude (but cute) version of this scheme based on the variable spelling of the word "smooch." The sender creates a template, indicating the locations where this emoticon may be placed (see Fig. 1.5). Then for each byte of the ciphertext, the encoder creates an instance of the word "smooch!" with a particular number of occurrences of each letter. The resulting message is sent to the recipient, who may decode it without requiring any additional information.

<> Hello Honey!

I hope you are having a nice morning. <> <>
(you can tell that I miss you -- I just wanna <> you all day)

<> <> (see?)

Oh, the box of chocolates you sent arrived this morning. <>
You're ever so thoughtful! <> I'm hoping you receive the
sweater I knit for you either in today's mail or tomorrow's.

Gosh I wish you were hear right now so I could <> your face!
<> <> <>
When did you say you would be done with that stupid conference? (<>)

Well, I've got a busy day ahead of me. I hope I survive it! <>
Thinking of you constantly will certainly help.

I'll be accumulating kisses in your absence -- so don't be surprised if you walk through
the door and get attacked by <> <>

Missing you desperately,

-- The <> Monster

p.s. <> <> <>

Figure 1.5. Example template used for smooch-coding

The source code for the encoder, "ENSMOOCH.C," and the decoder, "DESMOOCH.C," are given in Sections A.1 and A.2. A sample template file is shown in Fig. 1.5 (with the smooch locations specified by "<>"), and Fig. 1.6 shows the stegotext output produced by the encoder for this template.

The expansion ratio of this scheme is roughly 40:1, which is probably not acceptable. The work required by the sender for each message is also exorbitant. And it could be argued that the attacker is likely to be so nauseated by the stegotext that the communicating parties will be persecuted in any case. This scheme is interesting primarily for its illustrative qualities (or entertainment value) rather than its security.

SMOOOOOOCH!! Hello Honey!

I hope you are having a nice morning. SSSMMMOOCH!!! SSMMMOOOOOCH
(you can tell that I miss you -- I just wanna SSSMMMMOOOOOOCH you all day)

SSSSMMMMOOOCH!!! SSSMMOOOOCH!! (see?)

Oh, the box of chocolates you sent arrived this morning. SMOOOOOOOOOCH
You're ever so thoughtful! SSSMMMMOOOCH! I'm hoping you receive the
sweater I knit for you either in today's mail or tomorrow's.

Gosh I wish you were hear right now so I could SSSSMMMOOOOOOCH!!! your face!
SSSMMMOOOOOOOOOCH!! SSMMOOOOOOOCH!!! SMMMOOCH!
When did you say you would be done with that stupid conference? (SSSMMMMOOOOOCH!!!)

Well, I've got a busy day ahead of me. I hope I survive it! SSSMMOOOOCH
Thinking of you constantly will certainly help.

I'll be accumulating kisses in your absence -- so don't be surprised if you walk through
the door and get attacked by SMMMMOOCH! SSSSMMOOOOOOCH!!!

Missing you desperately,

-- The SMMOOOOOCH!! Monster

p.s. SSSMMMMOOOOOOOOCH!! SSSSMMMMOOOCH!! SSSMMMMOOOOOOOCH

Figure 1.6. Output of encoder with hidden message "Please return my albums."

1.3.1.4 Texto

Texto, by K. Maher [9], is a rudimentary text steganography program which transforms ASCII data into simple English sentences and back again. Since ciphertext can be stored in ASCII format (using uuencode or PGP's ASCII-armoured mode, for example), this system may be used to hide encrypted data in English text.

The natural language generation basis of Texto is similar to that of the bonzo-coding schemes presented here, only it is much less general. The encoder uses a file containing specifications of several sentence structures (templates) along with lists of 64 words of each of the following types: verbs, adjectives, adverbs, places, and things. The sentence templates are of the form shown in Fig. 1.7.

```
"The _THING _ADVERB _VERBs to the _ADJECTIVE _PLACE."

"Shall we _VERB before the _ADJECTIVE _THINGs _VERB?"
```

Figure 1.7. Sample template sentence structures for Texto

During the encoding process, a template is chosen (sequentially from the structures file) and each of the placeholders is replaced by one of the words in the corresponding list. The word chosen from the list depends on the value of the next six bits of the ciphertext stream. For example, given structures above and the input ciphertext stream "n_1 n_2 n_3 ..." (where each n is a six-bit number), it would insert the n_1-th word in the "THING" wordlist into the spot marked by _THING. The n_2-th adverb would take the place of the first "_ADVERB", and so on. The output would look something like the text shown in Fig. 1.8.

```
"The can lustily restrains to the quick swamp."

"Shall we sniff before the opaque balls slide?"
```

Figure 1.8. Sample output of Texto

To transform this stegotext back into the original ciphertext, the decoder scans the stegotext for words which are present in its word lists (the same lists used during encoding). When it finds such a word, it outputs the 6-bit number corresponding to that word's place in its list. All of these 6-bit numbers are concatenated to form the original ciphertext file.

While the output of Texto can look similar to human-generated text, it suffers from a number of limitations. The most serious drawback is that the sender and receiver must both share the same word lists. Since it would be infeasible to change lists for every communication, the use of this encoding scheme would be rather easily detected by looking for repeated use of words of each predefined type (verbs, adjectives, adverbs, places, and things). The fixed 6-bit per word conversion rule is also somewhat limiting. Both of these weaknesses are addressed by the bonzo-coding schemes.

1.3.2 Mimic functions

The most academically sound work done to date in the area of text steganography is found in Peter Wayner's papers on "mimic functions." These functions use Huffman compression and expansion functions to reversibly transform one random sequence so that it has a distribution approximating that of another sequence. Wayner shows in [10] that these transformations are optimal in the sense that they produce the shortest length sequences with the desired distribution. As mimic functions are part of one of the bonzo-coding designs, we briefly describe them now.

We wish to produce Y as a function of random sequence X (the ciphertext) such that Y has the one-dimensional distribution of random sequence H, the characters in human-generated text. Let $f_H()$ be a Huffman compression function which assigns variable-length binary strings to the fixed-length binary strings of its input, using a Huffman tree generated from the probability distribution of H. Similarly, let $g_H()$ be the corresponding Huffman decompression function which uses the same Huffman tree to recover fixed-length binary strings corresponding to the variable-length binary strings of its input. Wayner shows that if X is an independent, uniformly distributed random sequence, and we compute

$$Y = g_H(X) \qquad (1.1)$$

then Y will have approximately the distribution of H, with Prob[Y=y] equal to the negative power of two nearest to Prob[H=y]. Note that since $g_H()$ is an expansion function, Y will be longer than X. To recover X from Y, we use the Huffman compression function, setting

$$X = f_H(Y) . \qquad (1.2)$$

Without a great deal of difficulty, we can extend this scheme so that Y has an n-dimensional distribution approximating that of H.

Wayner describes two ways of creating "mimic functions" based on these transformations: one is character-based and the other word-based. In the character-based scheme, Y is a sequence of n-grams (character sequence of length n) which has approximately the n-dimensional distribution of the human-generated text analyzed. As shown by Shannon as early as 1951 [11] and demonstrated in [12] and [13], among others, randomly generating n-grams in this way can provide an interesting and remarkably accurate imitation of the text used in the statistical analysis stage. The occurrence of nonsense words and the nongrammatical sentences will make the computer-generated nature of Y obvious to the human observer. However, this type of mimic function will effectively fool automated detection attacks based solely on the n-dimensional distribution of characters in text.

The second type of mimic function involves generating words from a Context-Free Grammar (CFG). A CFG is a system of production rules and literal strings, and its syntax may be like the example shown in Fig. 1.9. In this case, a PET is a variable which may take the value of either "dog" or "cat." The alternatives available for each production rule may be assigned a probability, as indicated by the numbers in parentheses.

```
Sentence -> "My" PET "has" PROBLEM "."

PET      -> "dog" (.5)  |  "cat"  (.5)

PROBLEM  -> "fleas" (.6)  |  "the flu" (.3)  | "rabies"  (.1)
```

Figure 1.9. Example Context-Free Grammar production rules

To use CFGs with mimic functions, we start with a CFG with all production rule alternatives assigned probabilities corresponding to their likelihood in human-generated text (this likelihood determined by fiat or analyzing a particular corpus). The alternatives are then arranged as a Huffman tree like the one shown below in Fig. 1.10.

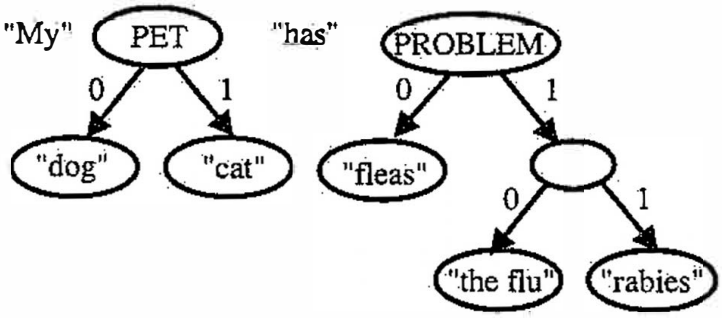

Figure 1.10. Huffman tree for the example CFG mimic function

Note if we generate a "Sentence" by making purely random choices at each of the branching nodes, we will have PET equal to "dog" half of the time and PET equal to "cat" the other half. Similarly, our pet will have "fleas" half of the time, and "the flu" and "rabies" each one fourth of the time. These probabilities are exactly those specified in the CFG production rules in the case of "cat" and "dog." The values of the variable PROBLEM occur with probabilities that are the negative power of two closest to the specified probabilities.

The steganographic encoding system involves using the bits of the input ciphertext to make the choices at all decision branches of the CFG. Assuming the Sentence defined above appears first in the stegotext, it will contain the word "dog" if the first bit is "0" and "cat" if this bit is "1." Since the ciphertext is assumed to be white noise, the word probabilities will approximately match those specified in the CFG production rules. To decode the stegotext the recipient must have access to the same CFG (and thus its corresponding Huffman tree). Then the decoding process becomes a task of parsing the CFG to see which choices were made in the encoding process. The bits of the ciphertext are thus revealed. Note that for an unambiguous encoding of the ciphertext, we require that the CFG be unambiguous. That is, for every possible string in the stegotext, there must only be one set of production rules that can generate it.

The output stegotext of the CFG-based mimic function can be quite human-looking. Assuming the probabilities assigned to the CFG production rule alternatives are accurate, the stegotext will have a word distribution approximating that of human-generated text. The expansion ratio is not easily computed, although the example in [10] suggest a ratio of roughly 80:1.

Wayner points out that the indetectability of the CFG-based mimic functions is hampered by the fact that output distributions are effectively rounded to the nearest negative power of two. However, the fundamental weakness of this scheme is that it requires the sender and receiver to share a large amount of information. These CFGs are specific to a particular domain of human writing (the corpus analyzed to produce the production rule probabilities), and they must be kept out of the hands of attackers. In [14], Wayner asserts that these mimic functions can be made "as secure as RSA or factoring Blum integers," but this is only true in the sense that it is difficult for the attacker to derive a CFG which would generate the observed stegotext. If the attacker somehow obtains the CFG used, the security is lost since the attacker may then readily identify stegotext.

1.4 Contribution of Bonzo-coding

The aim of the novel schemes presented in this thesis to enable text steganography that is practical and suitable for use on the Internet today. These systems achieve a level of indetectability comparable to that of CFG-based mimic functions while requiring a minimum of shared information held between sender and receiver. The bandwidth of these schemes is at least comparable and potentially far superior to that of mimic functions or other text-based steganographic systems surveyed.

We shall assume that the plaintext of the hidden messages sent with these schemes are fairly short (roughly between one and several lines of text). If larger data sets are to be sent using these systems, they will have to be broken into smaller chunks and sent using multiple bonzotext messages. Practically speaking, we consider the occasion to transmit large data sets innocuously to be far less frequent than the desire to exchange short hidden messages. Steganographic systems based on hiding information in image or audio data would probably be more appropriate for the frequent exchange of large data sets.

1.5 Organization of This Document

In the remainder of this document, we present a class of novel schemes for performing text steganography. Four designs are explained in Chapter 2, and the implementation of one of the designs is explained in Chapter 3. The results obtained from the implemented system are discussed in Chapter 4, including the discussion of several statistical analyses on the system output. Sample stegotext produced by the implemented system can be found in Sections C.1, C.2, and C.3. Finally, Chapter 5 offers a few conclusions drawn from this work and suggests directions for future work in this area.

CHAPTER 2. BONZO-CODING SYSTEMS

2.1 Bonzo-Coding Fundamentals

2.1.1 Coding model and message format

The steganographic encoder is called the "bonzifier" while the decoder is called the "debonzifier." The format of the bonzotext is the same for all bonzo-coding schemes. There is a "Prelude" which is a short section that carries no hidden data. This is followed by the "bonzoblock," the section containing all of the stegotext data. The document may contain text after the bonzoblock which is called the "Ending" section. The Prelude and Ending sections may be human-generated or output by the language generation system used to create the bonzoblock. All that we require are that a Prelude be present and that the start of the bonzoblock be identifiable by the debonzifier. There are many ways of implementing the bonzo-coding system such that this constraint is met (see Chapter 3).

Bonzo-coding makes use of functions that convert words, expressed as ASCII strings, to numerical values. When we do not care about the specific algorithm used for this conversion, we use the notation "f(w)" as the function assigning a numerical value to the word w. Usually, we are only interested in β bits of the result, where β is considered a constant for a particular instance of bonzotext (generally β is in the range of one to seven bits). This function must be a true function, in that we may only have one possible output for a particular input. However, $f(\cdot)$ may be a many-to-one, i.e., it is possible that $f(w_i) = f(w_j)$ for $i \neq j$.

We define a simple algorithm to be used to implement $f(\cdot)$. "h(w)" is a simple hash function which adds the characters together and takes the lower β bits of the result. $h(\cdot)$ ignores punctuation within a word, as well as the case of the first letter in the word. The reason for this is that the human sender may wish to change the formatting, punctuation, and some of the capitalization of the stegotext without altering its hidden data. Note that the output of $h(\cdot)$ will be the same for the words "tender loin" as "rented lion" since the sequence of the characters does not affect the value of the hash.

The words of the bonzoblock are chosen by the bonzifier such that their value under $f(\cdot)$ is the ciphertext data X. That is,

$$x_i = f(w_i) \,. \qquad (2.1)$$

where w_i is a particular word in the bonzoblock and x_i is the corresponding b bits of the ciphertext. All words in the bonzoblock that correspond to a value of X are called "coded words." In implementing the bonzo-coding schemes, we may wish to have additional information carried in the bonzoblock, most notably the length of the hidden ciphertext (when the decoder must stop decoding). The details of accomplishing this are discussed in Section 3.3. For now, it is safe to assume that the coded words contain only values of X.

The simplicity of the decoding algorithm is the main distinguishing characteristic of bonzo-coding. All that is necessary to recover the ciphertext is for the decoder to perform the calculation in Eq. (2.1) for each coded word in the bonzoblock. There is no parsing of Context-Free Grammars needed, even if a CFG was used as a part of the bonzification process. Thus, the shared information required between sender and recipient is minimized. In some of the systems presented below, the calculation of $f(\cdot)$ requires a shared passphrase. But this is not an unreasonable requirement and we may still consider the shared information minimal.

The price paid for the extreme simplicity of the debonzifier is that the bonzifier must not only produce bonzotext which *looks* human-generated, statistically and qualitatively; the coded words it selects must also have the correct value under $f(\cdot)$. We argue that these requirements are not mutually exclusive. In the systems presented here, the bonzifier is essentially a natural language generation engine which attempts to produce text which satisfies both of these constraints.

2.1.2 Notation

The following notation and definitions are used for the remainder of this document:

w	Any word as defined by a character string not containing spaces or carriage returns (it may contain punctuation such as ".", ";", """).
W_B	The word sequence of the bonzoblock, modeled as a random sequence.
humantext	Shorthand for "human-generated text."

W_H	A random sequence of words with distribution like that of humantext.
X	The ciphertext stream, modeled as a random sequence of independent, uniformly distributed random variables (a white noise sequence).
X_i	A particular bitstring of sequence X. X_i is a uniform random variable.
$x_1 \mid x_2 \mid x_3$	The concatenation of bitstrings x_1, x_2, and x_3. If each bitstring is four bits long, then this result is equivalent to $x_1 \cdot 2^8 + x_2 \cdot 2^4 + x_3$.
$w_1 \mid w_2 \mid w_3$	The word concatenation of ASCII strings w_1, w_2, and w_3 (which do not themselves contain spaces). These strings are separated by at least a space or carriage return. Commas and other punctuation are also allowed.
[opt]	An optional value, usually an ASCII string.
Prob[A | B]	The probability of event A given event B.
β	The bits-per-word value used for an encoding.
k	The alphabet size of each codeword, i.e., $k = 2^\beta$.
f(w)	A text-to-numeric transformation mapping word into a β-bit code value.
ρ	The padding value, with $0 \le \rho \le 7$; $\rho/8$ indicates the approximate number of noncoded words for each encoded word. If $\rho=0$ then all words in body are encodings.
Lx	The number of bytes in the ciphertext input file.
Lb	The number of coded words in the bonzoblock.
Lw	The total number of words in the bonzoblock.

2.2 System 1: Simple Word-hashing

2.2.1 Description

The first system is the simplest and also the most limited. The operations of the bonzifier and debonzifier are depicted in Fig. 2.1 and may be summarized by the following points:

- f(w) := h(w).
- Each word in the bonzoblock is a coded word.

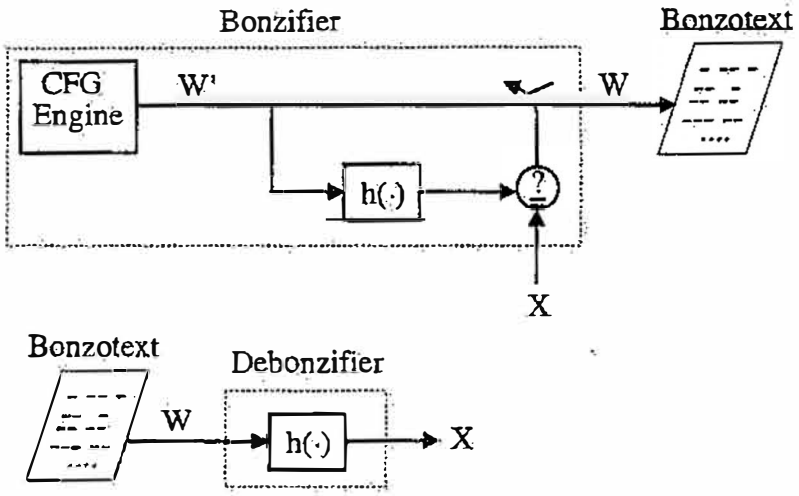

Figure 2.1. Block diagram of System 1

The bonzifier first breaks the ciphertext input into β-sized chunks, $x_1, x_2, ..., x_{Lb}$. Since Lb is the number of β-sized chunks needed to represent the Lx bytes of X, we have

$$Lb = \lfloor 8 \cdot Lx / \beta \rfloor, \qquad (2.2)$$

where $\lfloor y \rfloor$ is the least integer greater than or equal to the real-valued quantity y. For simplicity, consider the CFG engine as generating a sequence W' of candidate words. To determine if a particular candidate w is acceptable for the i-th position of the bonzoblock, the bonzifier simply checks if $h(w) = x_i$. This is a sort of "word filter" applied to the stream W' produced by the CFG engine.

In actuality, the bonzifier may be a tightly coupled combination of the natural language generator and the word filter. As a candidate linguistic structure (perhaps a phrase, sentence, or complete paragraph) is chosen from the language specified for it, each coded word location is checked. If,

for instance, the first few such words in a candidate sentence have the correct code values but the last one does not, the bonzifier may search for a word with the correct code value to replace the last word. In other words, it need not be an unintelligent generator that produces all possible linguistic structures and then look for ones with the proper coded words. This matter is discussed further in Section 3.6.

The task of the debonzifier is trivial. It simply computes Eq. (2.1) for each word w_i of the bonzoblock. It performs a binary concatenation of the β-bit values x_i to recover the original sequence X intact. Note that here we are assuming that the debonzifier knows Lx, the length of X in bytes, *a priori*. Section 3.3 discusses how this length information may, for convenience, be prepended to the ciphertext stream before encoding.

2.2.2 Analysis

This simple bonzo-coding system is primarily useful for illustrating the general scheme under investigation. It does achieve unambiguous textual encoding with minimal shared information, and the output bonzotext can look qualitatively like human-generated text if the CFG is carefully constructed.

As with all the bonzo-coding systems, the expansion ratio depends at least on the value of β used for a particular instance of bonzotext. If Lw is the average word length, then our expansion ratio is given by $(8 \cdot Lw / \beta) : 1$. Since the useful range of β was observed to be between 2 and 6, and assuming an average word length of five characters (a conservative estimate), we achieve expansion ratios roughly between 7:1 and 40:1. This ignores the characters of the Prelude and Ending sections of the bonzotext, as well as the punctuation between words of the bonzoblock. Therefore we may consider this a crude lower bound to the expansion ratio of this scheme. Nonetheless, the bandwidth achievable by this system is within usable limits; we can make use of a system that has 30 lines of bonzotext for a line of hidden text.

The computational efficiency is different for the sender and the recipient. The debonzification algorithm is a trivial computation, which runs in negligible time on personal computers and workstations. The time spent performing the decryption of the ciphertext is larger by at least an order of magnitude.

The time required by the bonzifier to encode a particular message might be much longer. If the CFG has too few alternatives for its production rules, then an error-free encoding may be either impossible or a slow computation. Experience to date has found this bonzification time to be anywhere from a couple of seconds to several minutes. These times are not considered unreasonable. The real implementation challenge is to make a CFG engine that does not require much "tweaking" on the part of the user (the sender) in order to attain an error-free encoding of the ciphertext. Such topics are discussed later in this document. Note that in this simple scheme, the value of h(w) may be precomputed by the bonzifier for all words w in a particular CFG.

This leads us to an interesting tradeoff involved in using any bonzo-coding scheme. If β is chosen to be low ($\beta = 1$ or 2), then it is much easier and quicker for the bonzifier to produce an error-free encoding using a particular CFG. But for low values of β, our expansion ratio will be fairly large, thereby lowering the achievable system bandwidth. On the other hand, if we use higher values of β, ($\beta = 7$ or 8), our bonzotext is much more compact but the bonzification time is longer and the chance of producing an error-free encoding from a given CFG is lower. In practice, producing a bonzotext with a particular CFG often requires trying a few different values of β.

Another factor that makes both the bonzifier's task difficult and the bonzotext less *human-looking* is that we require each word of the bonzoblock to carry a code value. There are solutions to this problem, one of which is presented in the design of subsequent bonzo-coding Systems.

2.2.3 Detectability

The hash function $h(\cdot)$ should ideally have independently distributed, uniform output when given the random sequence W_H. If this is so, then $h(W_H)$ will be statistically indistinguishable from $h(W_B)$ (which is the same as X) and the attacker will therefore not be able to distinguish between bonzotext and humantext based on the output of the debonzifier. However, the distribution of W_H is far from uniform and independent. This is due to the simple fact that humans do not use all words with equal frequency, and particular word sequences are far more likely than others (e.g., the pair "my money" is used far more often than the pair "my your"). So generating uniform, independent output of $h(W_H)$ will require using a very small number of bits (i.e., $\beta = 1$) and/or defining a different hash algorithm for $h(\cdot)$ which takes into account the distribution of W_H.

How distinct are the measurable distributions of $h(W_H)$ and $h(W_B)$ for the simple hash algorithm specified? Simulation results described in Section 4.4 indicate that they may be identified fairly reliably using Chi-squared tests or entropy measurements based on the one-dimensional distributions of $h(W_H)$ and $h(W_B)$ alone. Thus, the indetectability of System 1 is thought to be rather poor.

Since $h(\cdot)$ is permitted to be a many-to-one function, we can consider different hash algorithms which have a uniform output when given the input W_H. This is the approach taken in the design of System 4 (see Section 2.5). Accomplishing this can be difficult, so we next consider simple ways of divorcing the statistics of $f(\cdot)$ from the nonuniform distribution of W_H.

2.3 System 2: Word Padding and Hash Scrambling

2.3.1 Description

This system improves on the statistics of System 1 while requiring a small amount of additional shared information. It also frees us from the requirement that every word in the bonzoblock be a coded word. The system is shown in Fig. 2.2 and may be summarized by the following points:

- $f(w) := h(w) \oplus r$, where r is the value of a pseudo-random sequence.
- Word padding is used to allow for noncoded words in the bonzoblock.

In this diagram we see the addition of a new sequence, R. R is a sequence that is either one of several "constant" sequences known to sender, recipient and attacker, or it is the output of a pseudo-random number generator (PRNG) whose values are not known to the attacker. In the latter case, we may implement R using a cryptographically strong PRNG seeded with a passphrase shared by sender and recipient but unguessable by the attacker. The sequence R serves two purposes: it determines which of the words in the bonzoblock are coded words, and it "whitens" the output of the simple hash function $h(\cdot)$. We consider each of these roles in turn.

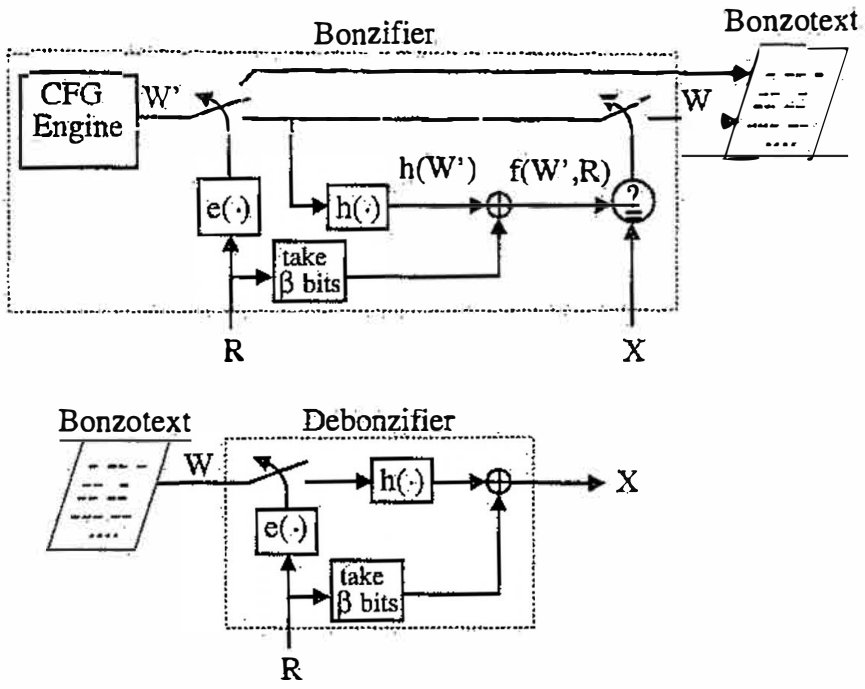

Figure 2.2. Block diagram of System 2

2.3.2 Word Padding

Allowing noncoded words to appear in the bonzoblock makes it easier for the bonzifier to come up with an error-free encoding which has the desired word statistics. It may be true that the attacker's job is complicated by not knowing exactly which words of the bonzotext represent code values, but the added security is not considered to be substantial. The sender specifies the amount of "word padding" to be used by giving a parameter ρ to the bonzifier. If ρ is equal to zero, then all words in the bonzoblock must be coded words. Higher values of ρ signify that more noncoded words should appear in the bonzoblock.

The bonzifier determines which words in the bonzoblock are coded words by processing the sequence R with a function e(·) that gives a yes/no binary output for each word in the bonzoblock. If we treat R as a white noise random sequence, then e(R) is simply a function that converts the independent, uniformly distributed bytes of R into a one-bit random variable which has Prob[no] proportional to ρ. We may implement e(·) as a simple look-up table.

Naturally, the debonzifier must know where the bonzifier put the coded words in the bonzoblock, so it performs the identical computation of e(R) using the same sequence R employed by the bonzifier. Thus R must be known to both the sender and receiver. If R is the output of the

PRNG described above, then this means that the communicating parties must share an unguessable passphrase. However, it is important to note that even if R is a nonrandom sequence, known to sender, recipient and attacker, it can still be useful to us. The constant sequence still provides a way to allow noncoded words in the bonzoblock. In Chapter 3, we see that R may be implemented as a choice of one of several constant sequences which are samples of a white noise variable. This obviously does not prevent the attacker from locating the coded words in the bonzoblock, but it does allow for better output of the bonzifier.

2.3.3 Hash Scrambling

The description above indicates that the word padding mechanism functions whether R is a true random sequence or a constant sequence. This is not so for the use of R to whiten the hash function output. We first consider the case in which R is a white noise sequence known to the sender and recipient but not to the attacker.

It is a well-known fact of cryptography that if you Exclusive-Or a random sequence of arbitrary distribution with a white noise random sequence, the output is another white noise sequence. This is the foundation for the "one-time pad" cryptosystem which is provably the *only* truly secure cipher [15]. Thus, if R is an independent, uniformly-distributed random sequence, and we compute

$$f(h(W),R) = \{ f(w_i,r_i) = h(w_i) \oplus r_i \}, \qquad (2.3)$$

then the output of $f(\cdot)$ is an independent, uniformly-distributed random sequence regardless of the distribution of $h(W)$. In the equation above, r_i is a β-bit chunk of the sequence R.

If R is a constant sequence, then there is no randomness involved in the calculation of $f(\cdot)$. If R consists of samples of a white noise sequence, then the bonzifier is likely to have an easier task of finding an error-free encoding of the ciphertext since the values of $f(\cdot)$ are more uniform. However, the detectability arguments of System 1 will apply. The attacker may easily recover the sequence $h(w_i)$ by either applying the hash function directly to the words or by computing

$$h(w_i) = f(h(w_i),r_i) \oplus r_i \qquad (2.4)$$

from the output of the debonzifier. Since R is not random, we will have $h(W_B)$ distributed as X which will be immediately distinguishable from the nonuniform $h(W_H)$.

2.3.4 Pseudo-code of Algorithms

To clarify the operation of System 2, the debonzifier algorithm is given in pseudo-code form in Fig. 2.3. Note that it is a simple, deterministic algorithm whose runtime is proportional to the length of X (or equivalently, the number of words in the bonzoblock).

```
for each word w in bonzoblock {
     let r = next byte of sequence R

     if e(r) = 0 then
           ignore w
     else
           next β bits of X = h(w) ⊕ (lower β bits of r)
}
```

Figure 2.3. Pseudo-code for debonzification algorithm of System 2

The pseudo-code for the bonzifier algorithm is given in Fig. 2.4. It is an over-simplified example of a bonzification algorithm which implements the word filtering stage and considers the CFG engine to be independent.

```
do {
     let x = next β bits of sequence X
     do {
           let r = next byte of sequence R
           get candidate word w from CFG
           if e(r) = 0 then {
                 output w to bonzoblock
                 found = 0
           } else {
                 if h(w) ⊕ (lower β bits of r) = x then  {
                       found = 1
                       output w to bonzoblock
                 }
           }
     } repeat until found=1 or all candidates tried
} repeat until no bits of X remain
```

Figure 2.4. Pseudo-code for bonzification algorithm of System 2

2.3.5 Analysis

The expansion ratios for bonzotexts produced by System 2 are increased by a function of the padding value ρ. In the implemented system discussed in Chapter 3, ρ takes a value between 0 and 7, where $\rho/8$ is the expected number of noncoded words for each coded word. Thus, the expected value of our crude lower bound on the expansion ratio is $(8 \cdot Lw / \beta) \cdot (1 + \rho/8) : 1$. For the implemented range of ρ, this yields expansion ratios between roughly 7:1 and 80:1.

The computational efficiency of System 2 is identical to that of System 1 in the case of the debonzifier, and slightly improved in the case of the bonzifier. The use of word padding effectively reduces the time required to produce an error-free bonzotext, as observed informally in experiments. This improvement in encoding speed is partially at the expense of an increased expansion ratio for larger values of ρ.

In System 1 the bonzotext is easily distinguished from the ciphertext because $h(W_B)$ necessarily has the distribution of X while $h(W_H)$ has a very nonuniform distribution. In System 2, we no longer require that $h(W_B)$ be distributed as X. If we can generate bonzotext that has W_B distributed the same as W_H, then we will clearly have $h(W_B)$ distributed the same as $h(W_H)$. Since $f(W,R) = h(W) \oplus R$, we know that $f(W,R)$ will be distributed as X regardless of the distribution of $h(W)$. Note that having W_B distributed as W_H is a daring assumption which directly implies that any statistical analysis will fail to distinguish between W_B and W_H.

We conclude that if the bonzotext is constructed with W_B distributed as W_H, and the sender and recipient share an unguessable passphrase, the attacker will not be able to distinguish between bonzotext and humantext. The question of *how* to generate bonzotext that has the proper code values and the distribution of humantext is addressed somewhat in Chapter 5 (using ideas from [16]). For now, we simply note that this is theoretically possible by the argument above.

It is natural to ask at this point whether or not we can perform indetectable bonzo-coding without requiring additional shared information (such as the passphrase). This is the aim of the next systems considered.

2.4 System 3: Mimic Function Processing of Ciphertext

2.4.1 Description

In the previous systems, we set $f(W_B) = X$ and attempted to make $f(W_H)$ look statistically like ciphertext X while enabling W_B to look statistically like W_H. Another way of achieving indetectability is to preprocess X, in a reversible manner, so that it has the statistical distribution of $f(W_H)$. Mimic functions provide a convenient way of accomplishing this.

Figure 2.5 shows the design of System 3. It is similar to System 1, with the following additions:

- We compute $Y = g_H(X)$ which has approximately the distribution of $h(W_H)$.
- Y is encoded using the simple hash function h(w).
- X is recovered from Y by computing $f_H(Y)$ after decoding the bonzotext.
- Word padding is used, as in System 2, with constant sequence R.

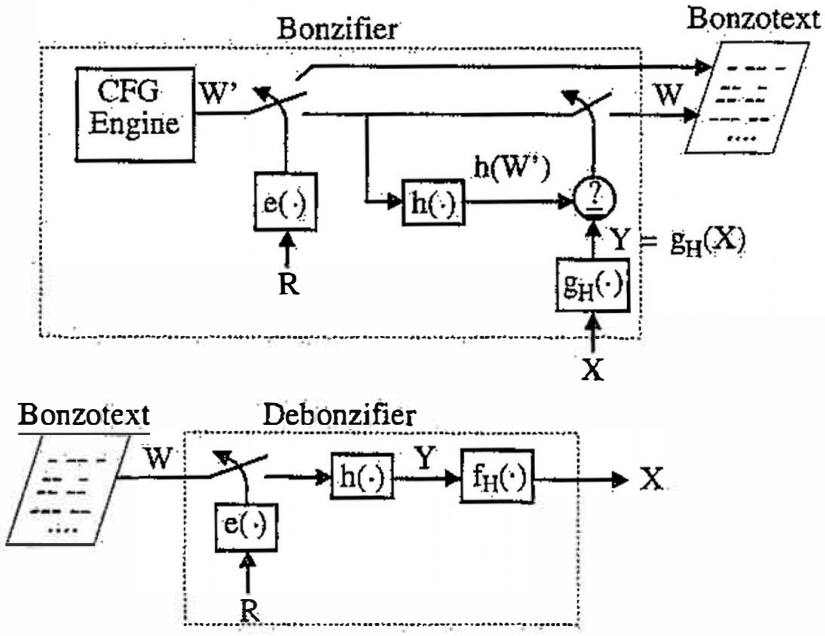

Figure 2.5. Block diagram of System 3

Before any bonzification can take place using this system, we first compute the n-dimensional distribution of $h(W_H)$ for a representative corpus of humantext. An appropriate value of n is to be found through experimentation. Using this distribution, we construct the corresponding Huffman trees in order to create the mimic functions $f_H(\cdot)$ and $g_H(\cdot)$.

The Huffman expansion function $g_H(\cdot)$ takes variable-length bitstrings and assigns them to fixed-length bitstrings using the Huffman tree based on the distribution of $h(W_H)$. In this case, the output bitstrings are $n \cdot \beta$ bits in length. As explained in Section 1.3.2, if X is a white noise random sequence, then $g_H(X)$ will have approximately the n-dimensional distribution of $h(W_H)$. We therefore wish to compute $Y = g_H(X)$ and use Y as the input to our simple bonzifier. Note that since $g_H(\cdot)$ is an expansion function, the length of Y will be longer than that of X (by a relatively small constant proportion).

The debonzifier first computes $Y = h(W_B)$ as in System 1 (with the word padding machinery added). Then to recover X, we simply compute $X = f_H(Y)$, where $f_H(\cdot)$ is the Huffman compression function using the same tree as $g_H(\cdot)$.

The word padding uses one of the known, constant sequences R to select the locations of the coded words in the bonzoblock. We do not require R to be a random sequence. The padding is only to make it easier for the bonzifier to produce an error-free encoding. Thus, we are not requiring any additional passphrases for this system.

2.4.2 Analysis

Since Y has approximately the n-dimensional distribution of $h(W_H)$, the output of the hash function should be approximately the same for $h(W_H)$ and $h(W_B)$. For a suitable choice of n, we will then also have that the output of $f_H(Y)$ statistically resembles white noise for both $Y=h(W_H)$ and $Y=h(W_B)$. Thus, the attacker will not be able to distinguish between bonzotext and humantext based on the observed statistics of Y or the statistics of the debonzifier output.

For complete indetectability, we also require that the distribution of W_B be sufficiently similar to W_H so that any statistical analysis of the text itself will also fail to distinguish between bonzotext and humantext. As with System 2, we have made it possible to satisfy this constraint, although we have not provided a direct means to do so.

The expansion ratio of System 3 is likely to be higher than the previous two systems, because we must expand X before encoding it as Y. This expansion, combined with the word padding, may stress the limits of acceptable expansion ratios. However, it is difficult to estimate the likely expansion ratios without knowing the appropriate ranges for β, ρ, and n for this system. It is possible that since Y is distributed as $h(W_H)$, the encoder may have an easier time finding word strings with the proper code values, as opposed to the task of finding such strings when the encoder input is ciphertext. Thus, we must implement and experiment with the system before knowing the useful values of β and ρ.

There are some other potential drawbacks to this system. The fact that $f_H(\cdot)$ and $g_H(\cdot)$ are specific to a particular body of analyzed humantext means that the distribution of our mimic functions will be somewhat erroneous when using a CFG corresponding to a different domain of humantext. Alternatively, we will require several versions of $f_H(\cdot)$ and $g_H(\cdot)$ to choose from, or we limit ourselves to a single domain of humantext. None of these options are terribly desirable.

The problem that the Huffman expansion function produces a sequence whose distribution is only accurate to the nearest negative power of two may also pose a problem. This might be fixed by using a different expansion/compression algorithm. Arithmetic compression [17] is one possibility and the use of homophonic ciphers [18] may also offer promise [16].

So while System 3 potentially offers improved indetectability without requiring passphrases, there must be additional development and experimentation applied to this design before its usefulness is known.

2.5 System 4: Compression of Hash Function Output

2.5.1 Description

Finally, we consider the use of word-to-number mappings other than the simple hash function specified. System 4 is depicted in Fig. 2.6, and its design may be summarized by the following points:

- A compression function is used to make $f(\cdot)$ distributed more like ciphertext.
- The candidate words are considered in sequences of length n.
- Word padding is used, as in Systems 2 and 3, with constant sequence R.

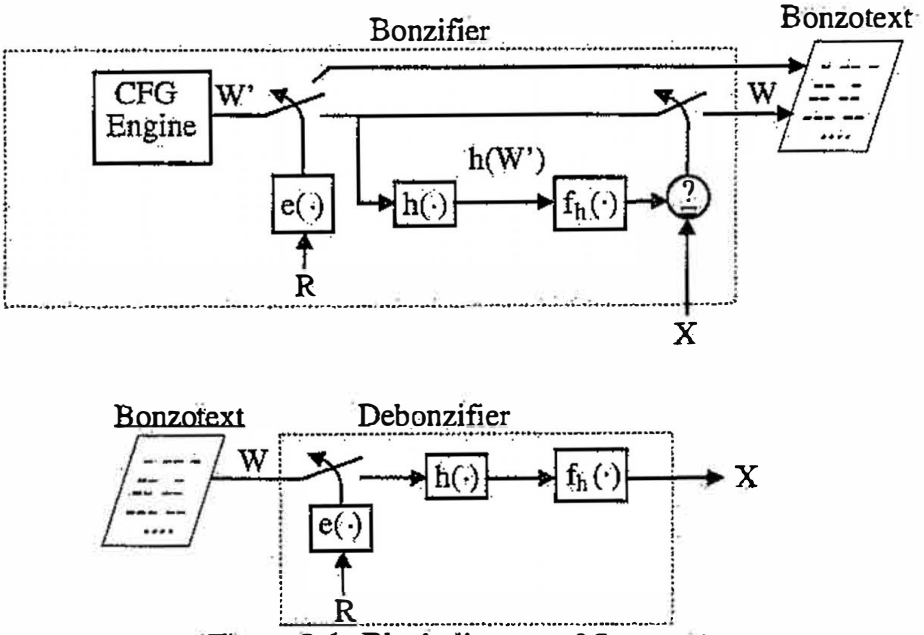

Figure 2.6. Block diagram of System 4

In Section 2.2, we found that by defining f(W) = h(W), the nonuniformities of W_H are clearly reflected in the sequence $h(W_H)$. One way of explaining this observation is that the output of our simple hash function $h(\cdot)$ is not sufficiently uniform with W_H as its input. We may attempt to correct this by employing a different hash function with more suitable output statistics.

A simple example which illustrates this technique is to process the output of the simple hash function $h(\cdot)$ with a compression function. Consider the Huffman compression function $f_h(\cdot)$, which uses the n-dimensional distribution of $h(W_H)$ to map $n \cdot \beta$-bit strings into variable-length bitstrings. The output of $f_h(h(W_H))$ will have a distribution substantially more uniform than $h(W_H)$ itself.

We make use of this in our bonzo-coding system by assigning

$$f(w_1, w_2, \ldots, w_n) = f_h(h(w_1), h(w_2), \ldots, h(w_n)), \qquad (2.5)$$

and this word-to-number computation is performed identically by both the encoder and decoder. The bonzifier ensures that $f(w_i, w_{i+1}, \ldots, w_{i+n})$ is equal to the appropriate bitstring of X for each sequence of coded words $w_i, w_{i+1}, \ldots, w_{i+n}$ chosen to appear in the bonzoblock. Similarly, the debonzifier computes $f(w_i, w_{i+1}, \ldots, w_{i+n})$ for each n-length sequence of coded words in the bonzoblock and performs a binary concatenation of the results to obtain the ciphertext X. Note that we do not need to use the corresponding Huffman expansion function for any reason. As in System 3, we may use the word padding system with constant sequence R in order to make the bonzifier's job easier.

2.5.2 Analysis

Assuming that W_H is distributed as W_B, as before, the indetectability of this system will be fairly respectable. We are basing our assessment also on the assumption that the compression function $f_h(\cdot)$ produces sufficiently uniform, independent-looking output when given the input h(W). It is possible that compression functions other than the Huffman variety might be better suited for satisfying this requirement.

The expansion ratio of System 3 is likely to be significantly higher than the other systems. This is because the average length of the output of $f_h(h(W))$ will be less than $n \cdot \beta$ bits, so the bonzifier requires more n-word sequences to encode a particular sequence X than do the previous systems. The resulting expansion factor is thought to be a fairly small constant. As with System 3, we may see a higher useful range for β, and/or lower values of ρ.

A likely drawback to this system is the complexity required of the bonzifier. To achieve decent output statistics, we must deal with candidate words in n-length sequences. The value of $f_h(w_i, w_{i+1}, \ldots, w_{i+n})$ must be computed by the bonzifier for each candidate sequence "on the fly," since computing this value for all possible n-length word sequences in the language database would be impractical. Thus, the encoding process will almost undoubtedly be quite slow. It may also be quite difficult to implement an appropriate natural language generation engine for this scheme, given that the candidate words must be checked in multiword sequences.

We note that System 4 has the same dependence on the analyzed text domain as System 3. If, for example, we attempt to generate poetry bonzotext using a compression function $f_h(\cdot)$ created based on the statistics of $h(W_c)$, where W_c is the random word sequence corresponding to cooking recipes, the output of $f_h(\cdot)$ may not be sufficiently uniform.

CHAPTER 3. SOFTWARE IMPLEMENTATION

3.1. System Overview

The experimental bonzo-coding software has undergone several revisions, and it continues to evolve as weaknesses are discovered and improvements devised. In its current state, it serves to demonstrate the feasibility of the class of text steganographic schemes proposed, and allow for continued experimentation. However, neither its security nor its user-friendliness are at an acceptable level for distribution at present.

The "proof of concept" implementation of the bonzo-coding scheme uses the design of System 2. The sequence R is implemented as the output of a weak random number generator whose seed σ is known to sender, recipient and attacker. Thus, we may consider R a sequence of known constants. However, since there are several possible seed values, R is actually one of several available constant sequences.

Figure 3.1 is a block diagram representation of the implemented system, which indicates its information flow. The plaintext is encrypted, and the ciphertext decrypted, using the Pretty Good Privacy (PGP) software [2]. Normally, the beginning of a PGP-enhanced file contains several recognizable "header" bytes which describe its format (it may be symmetrically encrypted, signed, the identifier of the asymmetric key used for signing, etc.). A program called "Stealth" [19] removes or restores the header bytes so that we may convert a PGP-enhanced file to a file of pure ciphertext, and vice-versa.

The bonzifier requires the sender to specify a "language" file containing the Context-Free Grammar information to be used. The sender must also choose the values of the parameters β, ρ, and σ. In practice, the sender usually tries a few different sets of these values to find a selection that produces an error-free encoding of minimal length for a particular language file.

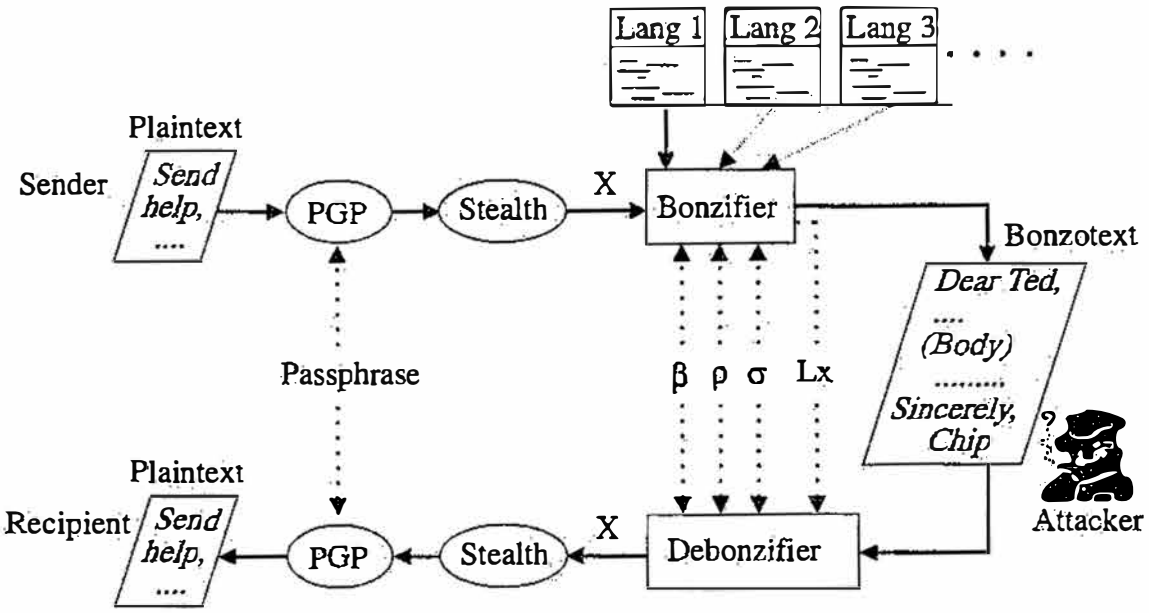

Figure 3.1. Block diagram of implemented system

The values of β, ρ, and σ, as well as the length of the ciphertext Lx, may each be encoded in the bonzotext itself or shared between sender and recipient via external channels. In the former case, the communicating parties maintain a minimum of shared information, thus making the system easier to use. In the latter, the bonzotext has fewer recognizable attributes and is therefore less detectable. We can imagine attacks where all text is scanned to see if the decoded output of the first few words of likely bonzoblock locations represents typical values of β, ρ, and σ, or if the length of the message is what would be expected if it were a bonzo-coding of Lx bytes of ciphertext. Although each of these four values may be included in, or excluded from, the bonzotext independently, we imagine that their inclusion will be governed by two principal ways of using the bonzo-coding software:

1. "Demo Mode"

   The users are not terribly concerned about maximal indetectability but are more interested in experimenting with the coding scheme and sharing bonzotext with a maximal audience. In this case, all parameters are encoded in the bonzotext.

2. "Paranoid Mode"

   If the communicating parties are indeed concerned about the system detectability, then they will probably exclude the parameters from the bonzotext and use external channels for sharing them. A reasonable special case of this is for two parties to agree to use the

same values of β, ρ, and σ for all bonzified communications. Then the value of Lx may either be included in the encoding (not a large degradation in security) or determined by such means as a unique "End-Of-File" symbol at the end of the ciphertext.

Each of the main steps involved in bonzo-coding is now explained in turn. Note that since Greek characters are not easily typed on standard terminals, the variables b, p and s will occasionally be used to represent β, ρ, and σ, respectively.

3.2 PGP Encryption, Decryption

As PGP is the current *de facto* standard software package for public-key cryptography, and it is available at no cost both within the United States and abroad, it is a natural choice for the cryptographic companion of the bonzo-coding software (or vice-versa). We make a few observations about how different ways of using PGP impact the performance of the bonzo-coding system.

To encrypt the plaintext data, the sender will either use a symmetric key (a passphrase) shared with the recipient, or she will use the recipient's public key. In the latter case, the need for a shared passphrase is avoided, as the data may only be decrypted using the recipient's corresponding private key. In either case, the plaintext is compressed by PGP before encryption. However, the encryption is performed on a block-by-block basis and additional information may be included in the ciphertext file. All of these have an effect on the resulting size of the ciphertext, and thus on the size of the bonzotext required to transmit it.

Typically, the compression operation is significant for plaintext messages longer than one line. If the resulting compressed data are longer than the original plaintext, the compression is bypassed. For short plaintext files, the length of the ciphertext is more dramatically affected by the choice of symmetric or asymmetric cryptography. If asymmetric cryptography is used, then the ciphertext is longer due to the fact that PGP symmetrically encrypts the plaintext with a random session key and precedes the resulting ciphertext with a key exchange block containing the session key encrypted under the recipient's public key. The observed results of choosing symmetric versus asymmetric encryption are summarized in Table 3.1. All values in the table are rough approximations. To minimize the length of the bonzotext, symmetric encryption was used for all experiments with the implemented bonzo-coding system.

Table 3.1. Approximate data length increase for PGP symmetric, asymmetric encryption

Plaintext Description	Plaintext Length	Data Expansion Ratio Symmetric	Asymmetric
A few words	10 bytes	5 : 1	15 : 1
A single line	50 bytes	2 : 1	4 : 1
A single paragraph	200 bytes	1.3 : 1	1.5 : 1

Processing the symmetrically encrypted data with Stealth removes the initial five bytes identifying the PGP version number, the format of the file (symmetrically encrypted) and related information. The output of Stealth is the ciphertext alone. Before decrypting the received ciphertext with PGP, the recipient must use Stealth to restore the information that was removed by the sender. Here we assume that the receiver knows *a priori* the type of encryption applied to the ciphertext, which is a reasonable requirement.

3.3 Bonzotext Structure

The structure of a bonzotext file used in the implemented system may be summarized as follows, using an informal notation similar to BNF.

- Bonzotext = Prelude, Body, [Ending]
- Prelude = Triggerword followed by a known number of lines
- Preamble = $[w_\beta]$, $[w_\rho]$, $[w_\sigma]$
- Lp = number of words in Preamble (Lp is between 0 and 3)
- Payload = $[Lx \mid] \; x_1 \mid x_2 \mid \ldots \mid x_{Lx}$
- x_1' = first β bits of Payload, x_2' = second β bits of Payload, etc.
- w_i = word chosen to encode x_i' (i.e., $f(w_i) = x_i'$)
- w_{nc}* = a sequence of noncoded words
- Body = Preamble $[w_{nc}^*] \; w_1 \; [w_{nc}^*] \; w_2 \; \ldots \; [w_{nc}^*] \; w_{Lw-Lp}$

We explain this format while referring to the example bonzotext in Fig. 3.2. The hidden message n this instance of bonzotext is "Destroy the tapes!" and it is unencrypted.

```
Dear William:
    I'm sorry I haven't written you sooner. I've been trying to come
up with a name for my grunge band.  Here are my best ideas so far:

"Gramma Judas & the censors"
"bastard scum"
"fish"
"David Mold & the ravenous breasts"
"heads"
"gooey torpedoes"
"the minstrel cramps"
"dang sisters"
"the critter"
"thirty-seven paperbacks"
"psychic jalopy"
"slug"
"twirling thirty-six boys"
"the wagon"
"angry king"

    Tell me if you have any favorites.
            Adam
```

Figure 3.2. Sample bonzotext with β=7, ρ=2

The "Triggerword" identifies the beginning of the Prelude, and its default value is the word "Dear." The use of a Triggerword allows bonzotext to be contained in a file with other text preceding it (mail headers, for example); we only require that the Triggerword not be used before the Prelude. The Prelude continues for a known number of lines before the Body of the bonzotext begins. As a default, the Body starts on the third line after the line containing the Triggerword. In Fig. 3.2, the Prelude consists of everything from the word "Dear" until the start of the grunge band names.

The Body is equivalent to the bonzoblock; it is the section that contains all coded words. The first words of the Body are the coded words carrying the values of β, ρ and σ, if the sender specified that these parameters be encoded in the bonzotext. These words constitute the "Preamble" and they are always coded with three bits per word. The recipient instructs the debonzifier as to what Preamble words exists in the bonzotext. For example, if the recipient tells the debonzifier that β=2, and that ρ and σ were encoded in the bonzotext (but not β), then the debonzifier will interpret the Preamble as "w_ρ w_σ." Note that even if the padding value ρ were chosen to be nonzero, the coded words of the preamble are never separated by noncoded words. In Fig. 3.2, the words "Gramma," "Judas," and "the" correspond to w_β, w_ρ, and w_σ.

The data to be coded consist of the ciphertext X, possibly preceded by the byte length of the ciphertext Lx. In the implemented system, 12 bits are used to encode Lx. These data are called the "Payload" and it is encoded in β-bit chunks. There may be noncoded words between the words carrying the Payload data. In a sense, the word padding is "turned on" after the Preamble. In Fig. 3.2, the Payload data are carried by the coded words between "censors" and "king," with the underlined words being noncoded words.

The bonzotext may contain an Ending section, consisting of words not used in the encoding. In Fig. 3.2 the Ending is the sign-off. Note that there is nothing to prevent the Prelude, Body, and Ending sections to be seemlessly integrated. Depending on the definition of the language file used by the bonzifier, the bonzotext may switch in midsentence from Prelude to Body, or Body to Ending.

3.4 Debonzifier Operation

The source code for the debonzifier, written in C, is given in Section A.3. It is a very simple program which can be easily made to run on any operating system platform of interest. The most complicated aspect of this program is that it implements its own pseudo-random number generator, since the R sequence must be the same as that generated by the bonzifier. A linear feedback shift register (LFSR) is used to generate this sequence. It is seeded by one of eight constants, as chosen by the value of σ.

The sequence R is simply the lower b bits of the LFSR output. To determine which words of the Payload section are coded words, the debonzifier maps the lower three bits of the LFSR output to a binary value. This is done using a lookup table which is constructed using the value of ρ. The table contains ρ zeros and $(8-\rho)$ ones. So when a uniform random variable is used as the index to the table, the table value will be zero (i.e., a non-coded word) with probability $\rho/8$. Thus, we will expect $\rho/8$ noncoded words for each coded word.

The steps taken by the debonzifier may be described as follows:

1. Accept any command-line arguments for β, ρ, σ, or Lx. If supplied, then assume these parameters will not be encoded in the Body.

2. Look for the first occurrence of the Triggerword in the input file. If not found, abort.

3. Find the start of the Body section by skipping the appropriate number lines after the Triggerword.

4. Using a bits-per-word value of 3, decode any Preamble words expected.

5. With word padding enabled and using β bits per word,

   5a. Decode enough words of the Payload to determine Lx (if it was not given as a command-line argument).

   5b. Decode enough words to output Lx bytes.

Note that the decoding performed in steps 5a and 5b follows the algorithm specified in Fig. 2.3, page 27.

3.5 Language File Specification

The language files specify the Context-Free Grammar to be used by the bonzifier. The language file given in Section B was used to create the example bonzotext shown in Fig. 3.2, page 38. We now briefly describe the syntax of the language files.

A production rule is defined by the "->" symbol. Alternatives for a particular rule are specified either by placing them on subsequent lines (before the next rule definition containing a "->"), or by separating strings on the same line with a "|" symbol. A single alternative may span multiple lines by ending the nonterminal lines with an underscore symbol. Probability weightings for production rule alternatives can be specified by including an integer in square brackets. For example, if a particular alternative contains "[2]," then it will be twice as likely to be chosen as alternatives with no probability weightings.

A terminal string is enclosed in double quotes. If the double-quote character is itself to be used in a terminal string, it appears twice in a row (note that we do not allow zero-length terminal strings).

The name of a production rule is essentially arbitrary, though it must not contain spaces. There are a few reserved names that must be defined in the language file. These names are Prelude, Body, Ending, and "ConsoleEcho." The meanings of the first three are obvious. The ConsoleEcho rule is a list which tells the bonzifier when it should display the output of a completed rule. For instance, the sender may wish to see each line of poetry as it is constructed, rather than wait for the entire output file to be completed before viewing it.

The pound symbol "#" at the beginning of a name has a special meaning. This tells the bonzifier to repeatedly evaluate the designated production rule until all of the data are encoded. Note that if the bonzifier runs out of data to encode midway through the evaluation of a particular production rule, it finishes evaluating the rule.

3.6 Bonzifier Operation

The source code implementing the bonzifier, written in language PowerBasic (for PC compatibles), is given in Section A.4. While the bonzifier source code is rather elaborate, its function is conceptually simple. It recursively searches through the production rules of the CFG to find a sequence of words which has the proper code values, using a variation of a depth-first search algorithm. Most of its complexity resides in its "MakeThing" subroutine, which is called recursively to evaluate production rules. When MakeThing finds a terminal string while evaluating the Body, it checks if the corresponding code values match the desired results (the bits of the Body) for the current word position in question.

The bonzifier implements the same LFSR and $e(\cdot)$ function as the debonzifier; however, these numbers may be pre-computed. In other words, the positions of all coded words with respect to the beginning of the Body section are determined before the search begins.

The main steps performed by the bonzifier are as follows:

1. Load the input file data into an array. Insert β, ρ, σ, and Lx at the beginning of this array, if they are to be encoded.

2. Load the information from the language file into a series of string arrays (unparsed).

3. Process the language information by parsing the production rules and pre-computing $h(w)$ for all words in all terminal strings.

4. Make the Prelude (with encoding turned off).

5. Make the Body (with encoding turned on).

6. Make the Ending (with encoding turned off).

When MakeThing evaluates a production rule with encoding turned off, it makes random choices whenever alternatives are available.

The search algorithm implemented by MakeThing works as follows. First note that in addition to returning a string representing the fully evaluated production rule, MakeThing also returns the number of coding errors associated with this evaluation. When evaluating a particular production rule for possible inclusion in the bonzoblock, MakeThing first picks a random alternative and begins to evaluate all of the rules it contains. If one of these rules is a terminal string, its coding errors may be calculated directly. If the rule is nonterminal, it must make a recursive call to evaluate that rule. Once MakeThing encounters errors while expanding and evaluating a particular alternative, it attempts to switch to another alternative. If expanding the new alternative begins to accumulate more errors than the previous alternative, it tries to find another alternative. If none exists, it goes back to expanding the alternative that accumulates the fewest number of errors. It continues this until a full evaluation is complete.

The evaluation returned by MakeThing is necessarily one with the fewest possible errors for that rule. This is a consequence of the strategy of continuing to expand the alternative with the fewest errors, switching alternatives whenever the one it is expanding gains more errors than another. In practice, we are only interested in error-free encodings, so the search may be expediated by discarding an alternative as soon as it is found to yield a single error. For purposes of experimentation, the implemented algorithm continues to consider such alternatives so that it may find a complete evaluation with the fewest errors even when an error-free evaluation is not achievable. It is important to note that the evaluation returned is not the *only* possible evaluation achieving the minimum number of errors. In this sense, the logic of MakeThing implements a "greedy algorithm"; it goes with the first optimal evaluation it finds.

The greedy strategy here implies that the bonzifier will not consider all possible alternatives for evaluating a rule. For example, let's say that the language file defines the "Body" rule as

```
Body -> Greeting Apology Excuses #Whats-up
```

and MakeThing produces an error-free evaluation of Greeting that is 15 words in length. It will then continue and attempt to find an error-free evaluation of Apology. It is possible that a ten-word Greeting would make an error-free Apology possible, whereas a 15-word Greeting positions the coded words such that all possible expansions of Apology produce errors. As currently implemented, MakeThing will never "back up" and re-evaluate a rule for which it already achieved a minimal evaluation. In this sense, it is a fairly naive search mechanism.

CHAPTER 4. RESULTS

4.1 Bonzotext Suitability

How do we measure the qualitative effectiveness of text-based steganographic systems? A truly successful bonzo-coding scheme might, for example, be able to create a Master's thesis that encoded a secret recipe for key lime pie, the plaintext being only available to the holders of a particular decryption key. The author refuses to admit whether or not this document is such an example. However, it is true that a number of messages generated using the fairly unsophisticated coding system described in Chapter 3 have been included in private communications and mailing list postings without their computer-generated nature being detected. More work will be needed to overcome Lincoln's Lemma[4] and achieve consistent indetectability.

It was found that the key to making the bonzotext unnoticeably different from humantext lies in the choice of domain of humantext writing used for the bonzotext generation. In a domain ideally suited for bonzo-coding, the human conventions governing what text is acceptable are either trivially concise (and thus easily implemented as a CFG) or extremely subjective and flexible (as with certain forms of poetry). There appear to be many "small" domains of writing that fulfill these criteria; examples include modern poetry, grocery lists, and names of grunge bands. As a result of experimentation to date, we assert that the generation of realistic bonzotext in such domains is far easier than attempting to produce more complex writing structures, such as complete letters.

In practice, it was determined that the embedding of bonzotext within humantext messages can give results that are rather inconspicuous. The casual eavesdroppers, modeled by mailing list observers in some experiments, tend to skim the contents of a message once the initial text leads them to believe that there is nothing unusual (or interesting) contained in the document. Thus, it is considered a strength of the bonzo-coding schemes presented here that the bonzotext may be embedded within humantext somewhat seemlessly.

In addition to the bonzotext shown in Fig. 3.2, three more examples of bonzotext are given in Sections C.1, C.2 and C.3. Although the CFGs for each of these examples are structurally simple and were constructed in a fairly haphazard manner, the bonzotext approaches a level of realism

[4] "You can fool some of the people all of the time... ," Abraham Lincoln, 1858 [20].

sufficient to fool the casual human observer. The recipe example in Section C.3 suggests that the context-free nature of the natural language generation system used will limit the level of realism achievable by this system (i.e., the choice of ingredients do not match the ingredients used in the preparation instructions). However, the complaint letter example in Section C.1 indicates that with the proper definition of the language file, some context may be established and kept consistent (in this case, the name of the target of the complaint).

4.2 Typical Expansion Ratios

The usable values of β and ρ varied with the choice of the language file. For CFGs consisting of simple lists with many alternatives available for its components, error-free encodings were obtained using $\beta=6$ or $\beta=7$ coupled with $\rho=0$ or $\rho=1$. This yielded an expansion ratio of roughly 20 bytes of bonzotext for every byte of ciphertext. Since the use of encryption for short messages accounted for an approximate 2:1 expansion, the overall expansion ratio was roughly 40:1. This means that for one line of plaintext, we would typically obtain a short page worth of bonzotext.

Other CFGs, such as those used to generate the complaint letter and poetry examples in Sections C.1 and C.2, contained longer fragments of human-generated text as string literals. The achievable values of β in these cases were found to be much lower (between 1 and 3), with the higher values of β requiring ρ values greater than one. Thus, the bonzotext is roughly 40 times longer than the ciphertext, and the overall expansion ratio is in the neighborhood of 80:1. As the poetry example in Section C.2 demonstrates, we can therefore get a full page of bonzotext for only a few words of plaintext.

In general, the word padding feature was found to be quite useful. In a previous version of the bonzifier, this option was not available and it was often difficult to obtain bonzotext that was both error-free and reasonably sized. Typically, the bonzifier could not find an error free encoding for one value of β, but the error-free bonzotext generated using the next lower value of β was unnecessarily long. Allowing for non-coded words in the bonzotext provides a greater continuum in parameter space, which improves the overall quality of the bonzifier output.

4.3 Time Efficiency

The debonzifier was observed to require negligible execution time for the short messages considered here. Since the debonzification time was clearly less than one half of a second on all hardware platforms considered, the actual time was not measured in the experiments on the implemented system.

The time required by the bonzifier to produce an error-free encoding also varied substantially with the choice of language files. Simple CFGs, with relatively few production rules and literal strings, would typically lead to bonzification times under three seconds when run on a 486-based, 50 MHz PC system. CFGs with more alternatives (such as the poetry language) would require up to five minutes to complete a bonzotext. Overall, bonzification times between 10 seconds and one minute were mostly frequently encountered.

The above are only measurements of the time required to produce a single instance of bonzotext. To find the appropriate β and ρ values for a particular language file, anywhere between three and ten trials were required. This process would only have to be done once for a particular language file (after which, several messages may be sent using it), and the steps involved are simple enough that this optimization could be automated. Once the appropriate β and ρ values were found, several trials were also needed to produce an optimally realistic bonzotext for a particular message. This step required roughly between three and five bonzifications. This per-message filtering could be omitted if the language file is known to regularly produce acceptable bonzotext.

The most time-consuming step of using the bonzifier, as implemented, was that of preparing a few useful language files. Some of them were constructed "from scratch"; this is a laborious yet educational process which could not be expected of every sender wishing to use the bonzo-coding system. However, the proliferation of natural language generation engines on the Internet has made feasible the task of modifying an already functional system so that it works with or as the bonzifier. For example, using the source code from Pakin's "Complaint Generator," [21], [22] a corresponding language file was constructed within a couple of hours. The adaptation process mainly consisted of a series of string search-and-replace operations. This experience leads us to the assertion that any number of natural language generators available today could be modified to work in the bonzo-coding scheme without much difficulty.

4.4 Statistical Characteristics

The problem facing anyone designing a cryptographic system for practical use is that it is impossible to prove that the system is impervious to all forms of analysis and attack (the one-time pad is the only exception). Furthermore, it is impossible to anticipate all attacks that could be used against the system. The best that can be done is to try as many attack strategies as possible and see if any succeed. Similarly, we cannot truly show that a bonzo-coding system or its implementation can provide indetectability against all forms of analysis that an eavesdropper might employ. Thus, we perform a few simple tests using standard statistical methods now and apply more "torture testing" to the system as its development continues.

For the purposes of obtaining initial measurements of the suitability of the implemented system, two types of statistical tests were performed. Both are measurements based on the one-dimensional frequency distribution of the sequences in question. That is, we model the sequence as samples of a random variable and estimate the one-dimensional probability distribution of that random variable using the relative frequency of each of its output values.

The statistical tests were performed using five sequences defined as follows:

- "random" is a sequence of random numbers from the system "rand(·)" function
- "ciphertext" is a sequence of data symmetrically encrypted using PGP and Stealth
- "debonzo(humantext)" is the output of the implemented debonzifier when given humantext samples as input
- "hash_only(humantext)" is the output of h(·) when given humantext samples
- "huffman" is the huffman encoding of the humantext samples

The "random" and "huffman" sequences were included mainly for reference. The primary interests are the similarities and differences between the other three sequences.

For each sequence, five measurements were taken at each of four different sequence lengths. The sequence lengths correspond to typical lengths of bonzotext messages, and each measurement is taken using a different set of humantext files as input. The humantext was chosen from online newspapers as well as personal correspondence.

4.4.1 Entropy test

The entropy of a random variable can be thought of a measure of its randomness or information content. It is computed as the sum of $p_x \cdot \log_2(1/p_x)$ for all probabilities p_x. A uniformly distributed random variable has the highest entropy possible. Figure 4.1 is a plot of the entropy measurements for our sequences of interest. Note the sequences are 8-bit values, and the entropy of an ideal, uniform random variable with 256 possible values is also eight bits.

Intuitively, we expect the "random" sequence to have the highest entropy, and indeed this is confirmed in measurements. The ciphertext should also have statistics similar to a random sequence, which is also observed in the tests. The fact that each of these sequences has an entropy somewhat less than eight bits is mainly due to the small length of the sequence; that is, nonuniformities develop in the distribution merely because there are not enough samples. The entropy approaches eight bits as the sequence length increases.

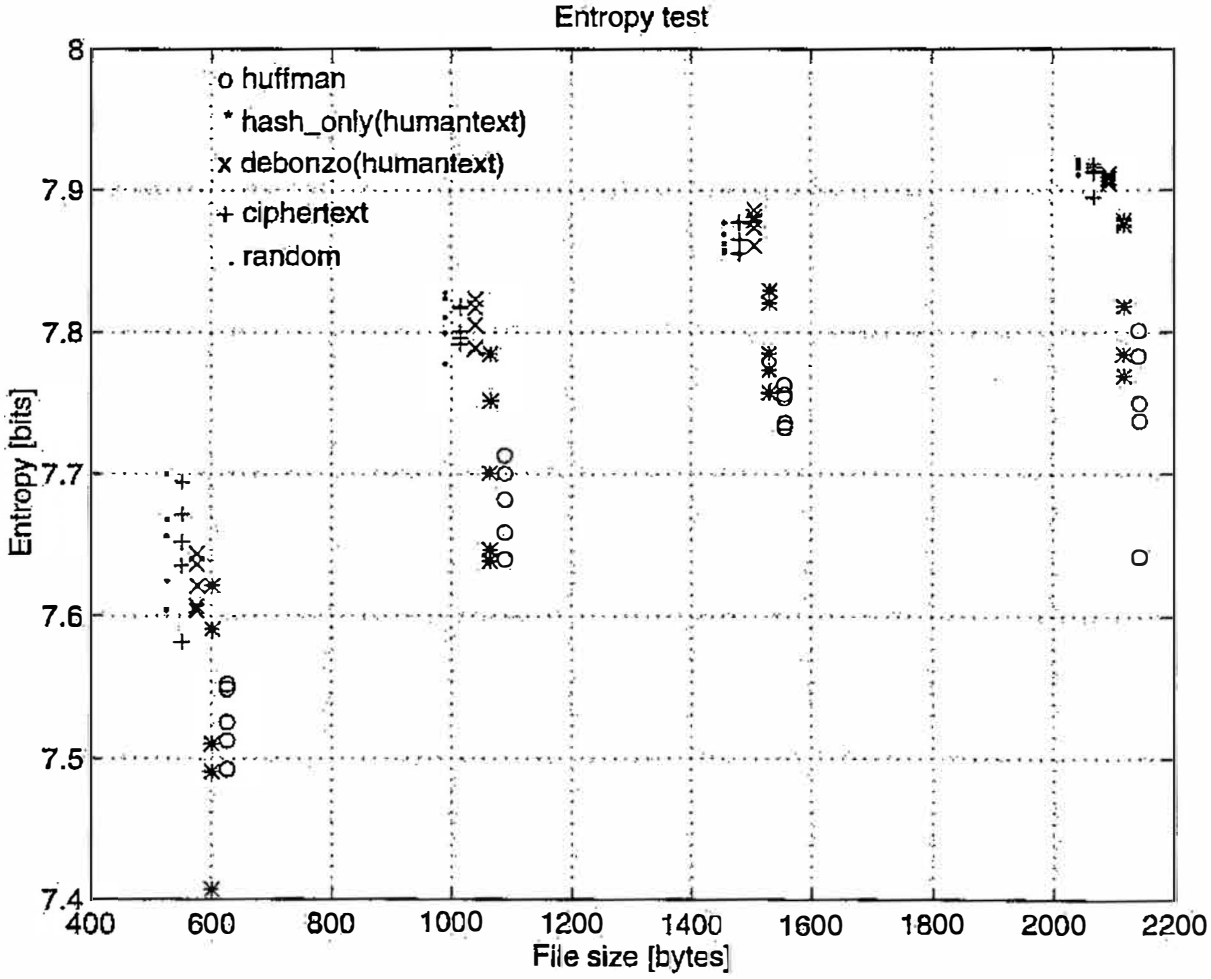

Figure 4.1. Entropy test

There are a few interesting observations to be made from Fig. 4.1. First, the hash_only output has a significantly lower entropy than the ciphertext, so much so that distinguishing between $h(W_H)$ and $h(W_B)$ is likely to be fairly reliable based on the entropy statistic. However, the output of the debonzifier in the implementation is $f(W) = h(W) \oplus R$. Since the implemented system uses constant sequences for R that are somewhat crude samples of a white noise random sequence, the debonzifier output can be considered a rough approximation of f(W) using a strong PRNG for R. In Fig. 4.1, the debonzifier output for humantext, $f(W_H)$, displays an entropy which is quite comparable to that of the ciphertext and also the "random" sequences. The low entropy of the Huffman-encoded humantext is probably due to the algorithm chosen for assigning ones and zeros to the branches of the Huffman tree. For example, if the branch corresponding to the node with the higher probability is always assigned a one, then ones will occur more frequently in the output of the Huffman encoder.

4.4.2 Chi-squared test

The Chi-squared test is a tool commonly used in statistical analyses to see how closely the measured distribution of one random variable matches that of another. Here we are mainly interested in how similar the distributions of hash_only(humantext) and debonzo(humantext) are to that of ciphertext. However, since we do not have a suitable theoretical model of the distribution of ciphertext (other than that of white noise), we may accomplish this by comparing the measured distributions of all five sequences to what would be expected from an ideal, uniform random variable. This is exactly the experiment described by Knuth [23] in measuring the "randomness" of a particular pseudo-random number generator.

Figure 4.2 shows the measured Chi-squared statistics using the byte-valued output of each of the sequences. Thus, the ideal random variable in question has 256 equally likely values. The Chi-squared statistic that compares two 256-valued variables is said to have 255 degrees of freedom. There are statistical tables which indicate how likely a particular value of the Chi-squared statistic is for an ideal random variable with the specified degrees of freedom. For example, a Chi-squared statistic calculated for an ideal, uniform random variable with 255 degrees of freedom is known to be greater than 252, 75% of the time, and greater than 286, 25% of the time. If the measured Chi-squared statistic for a random variable under test is within roughly the 15% to 85% range, then it is considered to match the distribution of the ideal random variable fairly well.

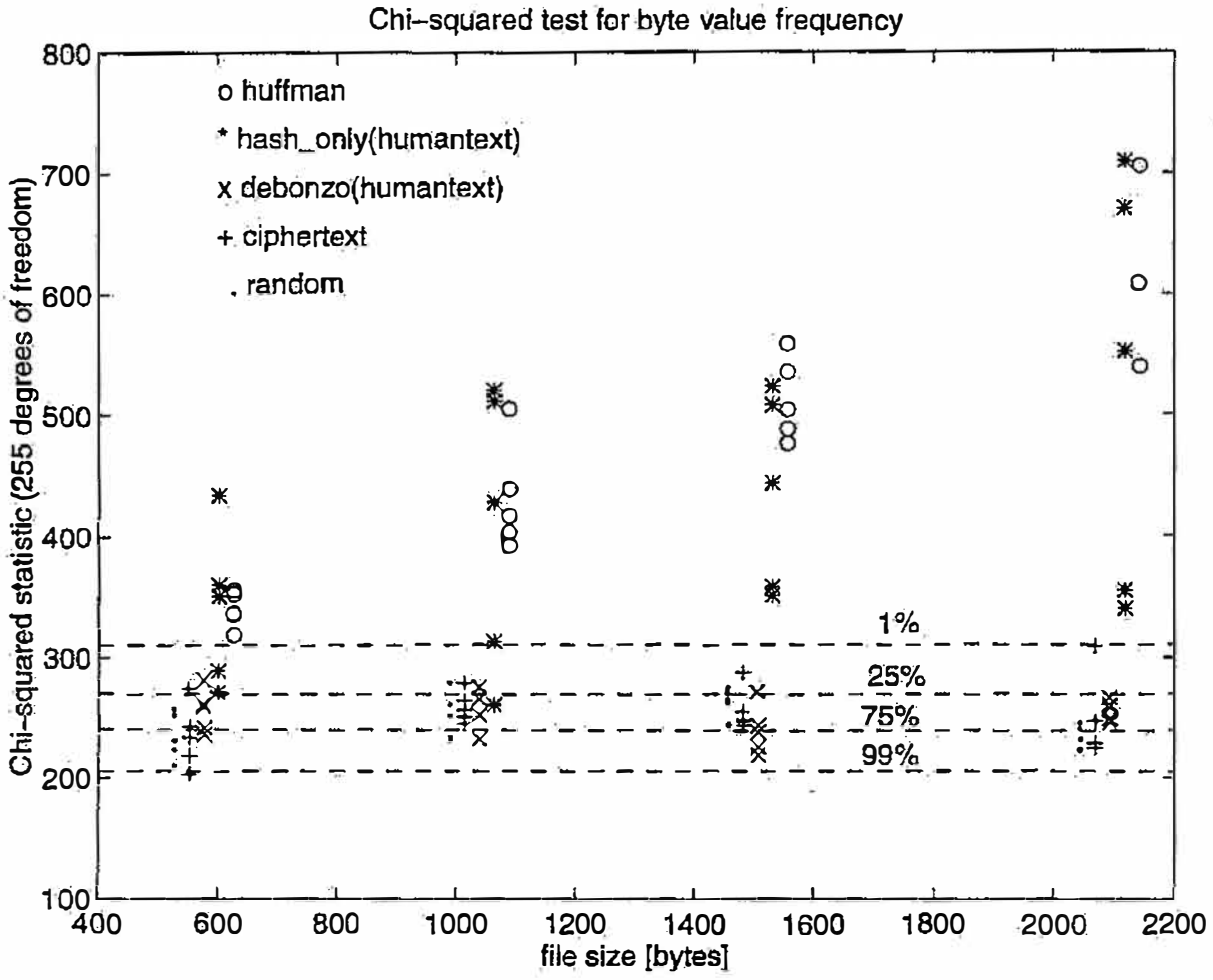

Figure 4.2. Chi-squared statistic for byte value

n Fig. 4.2, we see again that the "random" and "ciphertext" sequences display behavior quite imilar to the ideal uniform random variable. In addition, the debonzifier output (with humantext s input) is also in the same range, which is another indication that $f(W_H)$ is difficult to distinguish rom ciphertext for System 2. Note however that the hash_only output is very different from iphertext by this measure. Thus, the System 1 output $h(W_H)$ will be easily discerned when ompared with $h(W_B)$, the ciphertext.

inally, we may also measure the Chi-squared statistic using the frequency of bit values in each equence, as opposed to the byte values. Thus, we are comparing the measured distributions of ıese sequences with what we expect from a binary, uniformly distributed random variable, i.e., a ir coin toss. This involves measuring the Chi-squared statistic with one degree of freedom.

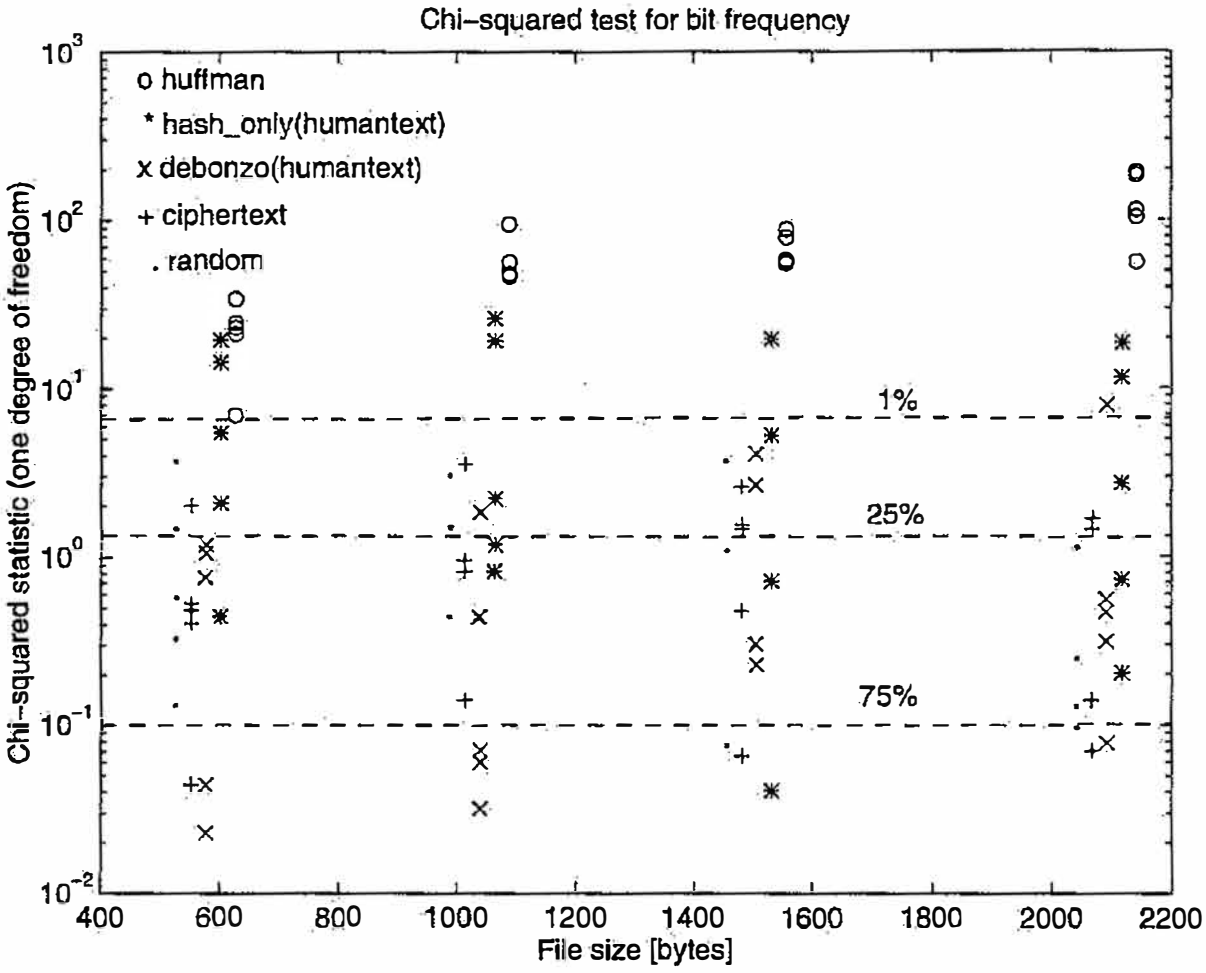

Figure 4.3. Chi-squared statistic for byte value

Figure 4.3 displays the results of this "bit frequency" Chi-squared test. Note that the vertical axis uses a logarithmic scale. Here again we see that the random and ciphertext sequences have fairly ideal statistics, and the debonzifier output is comparable though less well-matched than in previous tests. The observed imbalance of one and zero values in the Huffman-encoded data supports the assertion that the branches of the Huffman tree are not assigned code values in a sufficiently random fashion.

Naturally there are many other statistical tests that should be performed on such a system before a useful assessment of its security is at hand. Interestingly enough, several of the tests used in

cryptanalysis to determine if a candidate decryption is successful are also applicable to this task. The cryptanalyst also desires to quickly tell the difference between ciphertext (the output of a decryption with the wrong key) and non-random, possibly human-generated data (the correct decryption). Tests such as the Index of Coincidence and the variations on Sinkov's Statistic, described in [24] and [25], would be worth performing in the context of the bonzo-coding system.

CHAPTER 5. DISCUSSION

5.1 Conclusions

From both of the analyses in Chapter 2 and the experience gained implementing the "proof-of-concept" system, it is our conclusion that the class of bonzo-coding schemes proposed show promise in providing a means for practical text steganography. The implemented system offers a convincing argument that such systems are feasible, though there is much work to be done before a bonzo-coding software package will be deployable and usable on a widespread basis.

It is notable that the bonzo-coding systems presented here allow error-free recovery of the ciphertext from the bonzotext with a minimum of shared information maintained between sender and recipient. This is possible because the statistics of f(W) are made to be independent of the statistics of W. The main theoretical question left unanswered is how to make a bonzifier that ensures that the desired n-dimensional word distribution is achieved in the bonzotext. This could be accomplished by selecting candidate n-length word sequences (or longer) and base the decision to include them in the bonzotext on both the code values achieved and the desired distribution [16]. We may also envision schemes that preprocess larger word sequences so that all candidates are drawn from a data set known to have the desired distribution.

In addition to the theoretical analyses of the detectability of the bonzo-coding systems, it is important to consider the humantext domain of the bonzotext. Practically speaking, we need not require that the bonzifier be able to generate perfect dissertations; if it can produce simple messages such as shopping lists, love poems, or dream descriptions, then we can still consider the system usable. Furthermore, it is important to note that if the bandwidth requirements are low, then the bonzotext can be made quite realistic by the use of low values of β and high values of ρ. The example shown in Fig. 3.2 demonstrates that frequent words such as "the" are easily included in the bonzotext if noncoded words are allowed. We conjecture that for a reasonably designed language file, increased word padding will make the word distribution of the bonzotext more accurately match that of the relevant humantext.

5.2 Future Work

This research is likely to be continued by exploring multiple paths concurrently. To allow more interested parties to experiment with bonzo-coding, we seek to translate the bonzifier source code into portable C so that it may be used on any operating system platform desired. This will also have the effect of increasing the computational efficiency of the bonzifier, since programs

compiled using PowerBasic are almost invariably slower than those compiled in C. Expanding the user base of the bonzo-coding software will make the necessary improvements in the system design more evident. Already much has been learned by collaborators attempting to write language files and use the bonzo-coding system.

The next step for continuing the development of the implemented system is to include support for a cryptographically strong PRNG, so that the system's statistical properties are improved. There are a number of miscellaneous enhancements that can also be made. For instance, there should be ways of identifying the location of a bonzoblock other than looking for a particular Triggerword.

On a different path, it is considered worthwhile to attempt the implementation of System 3 and/or System 4. It is possible that the use of mimic functions, as specified in these designs, will greatly increase the observable statistics of the bonzo-coding system. We may also see an improvement in performance since the bonzifier may be able to find error-free encodings more easily for these systems.

Further statistical analyses are warranted, for the implemented system as well as any future implementations of Systems 3 or 4. This is another area that would benefit greatly from the involvement of others, particularly those experienced in statistical analysis of ciphertext and related matters.

Perhaps the most important area of future investigation is that of improving the natural language generation engine of the bonzifier. Given the quality and abundance of current research in the areas of Natural Language Generation and Natural Language Understanding, we consider it quite possible that the task of constructing a language file used in the bonzification process may be automated. It is easy to imagine a program which runs in the background, scans the abundance of human-generated text available on the Internet, and gathers linguistic structures to be used for bonzo-coding. In this way, language files might be continually created and discarded after being used once by the bonzifier.

This model of automated language file modification has been the vision from the start of the bonzo-coding work. It is undesirable to have either a static set of language files used by all senders, or to require senders to hand-edit the language files. We must, however, always allow the sender to specify their language to a fine degree. Indeed, the educational process involved in learning to abstract and regenerate human textual communication is perhaps as valuable as any additional privacy provided by the steganographic system.

APPENDIX A. PROGRAM LISTINGS

The source code for several of the text steganographic systems discussed in this thesis are given in Sections A.1 through A.4. The smooch-coding scheme described in Section 1.3.1.3 uses the "ENSMOOCH" encoder, whose source code is given in Section A.1, and the "DESMOOCH" decoder, whose source code is given in Section A.2. Section A.3 contains the source code for the debonzifier discussed in Section 3.4. Section A.4 briefly describes some distinctive features of the PowerBasic language, and lists the source code for the bonzifier discussed in Section 3.6.

A.1 ENSMOOCH.C

```
/*      S: 1-4   [2 bits]
        M: 1-4   [2 bits]
        O: 2-10  [3 bits]
        CH
        !: 0-3   [2 bits]
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define OUTBITS    7
#define LENWORDS   1              /* number of words for bitlength */
#define BADWORD   -1

FILE *infile,*dummyfile;
char *word,dummystring[128];
int  i,j,k,wordbits;
int  inbuf[10000], inbufptr, buflen, totalbits = 0;
int  INandmask, OUTandmask;

/*********************************************************************/
int load_infile(void)
{
  int shiftreg = 0, cbits = 0;
  char c;

  inbufptr = LENWORDS;

  while ((c = fgetc(infile)) != EOF) {
    shiftreg |= (c & OUTandmask);
    cbits += OUTBITS;

    if (cbits >= wordbits) {
       inbuf[inbufptr++] = (shiftreg >> (cbits - wordbits)) & INandmask;
       shiftreg &= ((1 << (cbits - wordbits)) - 1);
       cbits -= wordbits;
       totalbits += wordbits;
    }
    shiftreg = shiftreg << OUTBITS;
  }
  buflen = inbufptr;
                                  /* a wordbits-bit number      */
  inbuf[0] = totalbits / wordbits - 1; /* number of words to decode */
}
```

```
/*****************************************************************/
char *encode_word(wordnum)
   int wordnum;
{
   int i, n, s = 0;
   char tmpsmooch[32];

   n = ((wordnum >> 7) & 3) + 1;
   for (i = 0; i < n; i++)
      tmpsmooch[s++] = 'S';

   n = ((wordnum >> 5) & 3) + 1;
   for (i = 0; i < n; i++)
      tmpsmooch[s++] = 'M';

   n = ((wordnum >> 2) & 7) + 2;
   for (i = 0; i < n; i++)
      tmpsmooch[s++] = 'O';

   tmpsmooch[s++] = 'C';
   tmpsmooch[s++] = 'H';

   n = wordnum & 3;
   for (i = 0; i < n; i++)
      tmpsmooch[s++] = '!';

   tmpsmooch[s++] = '\0';

   printf("%s ",tmpsmooch);

   return tmpsmooch;
}
/*****************************************************************/
main(argc,argv)
   char* argv[];
   int argc;
{
   int wordnum, nbits = 0;

   if (argc < 3) {
      puts("USAGE: ensmooch infile dummyfile");
      exit(-1);
   }

   wordbits = 9;
   INandmask  = (1 << wordbits) - 1;
   OUTandmask = (1 << OUTBITS) - 1;

   if (!(infile = fopen(argv[1],"rt"))) {
      printf("Can't open input file: %s\n",argv[1]);
      exit(-1);
   }

   if (!(dummyfile = fopen(argv[2],"rt"))) {
      printf("Can't open dummy file: %s\n",argv[2]);
      exit(-1);
   }

   load_infile();
   fclose(infile);
   puts("");
   inbufptr = 0;

   while (fgets(dummystring,128,dummyfile)) {
     word = strtok(dummystring," ");
     while (word) {
       if (strncmp(word,"<>",2) == 0) {
```

```
         if (inbufptr < buflen)
            encode_word(inbuf[inbufptr++]);
         else
            encode_word(0);

         if (strstr(word,"\n") != NULL) {
            puts("");
         }
      }
      else {
         if (word[0] == 32)
            word++;
         printf("%s ",word);
      }
      word = strtok(NULL," ");
    }
  }
  fclose(dummyfile);
  puts("");
}
/*************************************************************************/
```

A.2 DESMOOCH.C

```
/*     S: 1-4   [2 bits]
       M: 1-4   [2 bits]
       O: 2-10  [3 bits]
       CH
       !: 0-3   [2 bits]
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define DEFAULTWORDBITS   9
#define OUTBITS    7
#define LENWORDS   1            /* number of words for bitlength */
#define BADWORD   -1

ILE *infile;
har word[128],dummystring[128];
nt  i,j,k,wordbits,nbits = 0;
/*************************************************************************/
nt decode_word(word)
  char* word;

  int length, i, ok = 0, c, ret = 0;

  length = strlen(word);
  for (i=0; i<length; i++)
     word[i] = toupper(word[i]);

  if ((strstr(word,"SM") == NULL) || (strstr(word,"OOCH") == NULL)) {
    return BADWORD;
  }

  i = strchr(word,'M') - strchr(word,'S');
  ret = (i-1) << 7;

  i = strchr(word,'O') - strchr(word,'M');
  ret += (i-1) << 5;
```

```
   i = strchr(word,'C') - strchr(word,'O');
   ret += (i-2) << 2;

   if (strchr(word,'!'))
      i = strchr(word,'\0') - strchr(word,'!');
   else
      i = 0;
   ret += i;

   return ret;
}
/*****************************************************************/
main(argc,argv)
   char* argv[];
   int argc;
{
   int shiftreg = 0, totalbits = 0, cbits = 0, wordnum, outch;
   int INandmask, OUTandmask;

   if (argc < 2) {
      puts("USAGE: desmooch smoochfile");
      exit(-1);
   }

   wordbits = DEFAULTWORDBITS;
   INandmask  = (1 << wordbits) - 1;
   OUTandmask = (1 << OUTBITS) - 1;

   if (!(infile = fopen(argv[1],"rt"))) {
      printf("Can't open input file: %s\n",argv[1]);
      exit(-1);
   }

   puts("");

   for (i = 0; i < LENWORDS; ) {
      fscanf(infile,"%s",word);
      if ((wordnum = decode_word(word)) != BADWORD) {
        nbits = wordnum * wordbits;
        i++;
      }
   }

   while ((fscanf(infile,"%s",word) != EOF) && (totalbits <= nbits))
     if ((wordnum = decode_word(word)) != BADWORD) {
           shiftreg |= (wordnum & INandmask);
           cbits += wordbits;


           while (cbits >= OUTBITS) {
              outch = (shiftreg >> (cbits - OUTBITS)) & OUTandmask;
              shiftreg &= ((1 << (cbits - OUTBITS)) - 1);
              cbits -= OUTBITS;
              putc(outch,stdout);
              totalbits += OUTBITS;
           }
           shiftreg = shiftreg << wordbits;
     }
   fclose(infile);
   puts("");
}
/*****************************************************************/
```

A.3 DEBONZO.C

```
/*   USAGE: "debonzo bonzofile [options]
 *
 *   options:
 *              -debug
 *              -o outputfile
 *
 *   options used in paranoid mode:     (used also during encoding)
 *
 *              -xlen   length of plaintext is n bytes
 *              -b n    use wordbits=n
 *              -p n    use padding=p
 *              -s n    use n as random seq generator seed
 *              -b n    bonzotext begins on n-th line after trigger word
 */

#define DOS

#ifdef DOS
#define STRICMP stricmp
#define STRNICMP strnicmp
#else  /* Unix */
#define STRICMP strcasecmp
#define STRNICMP strncasecmp
#endif

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define DEFAULTWORDBITS   2
#define OUTBITS    8
#define LENBITS   12          /* num bits used to code bytelength of payload  */
#define BADWORD   -1

char *triggerword = "Dear";
int  beginsentence = 3;       /* begin decoding this many lines after trigger  */

FILE *infile, *outfile;
char word[128], dummystring[128];
int  i, j, k, wordbits = 0, debug = 0, outf = 0;
int  xb = 0, xpad = 0, xseed = 0, xlen = 0;
unsigned int nbytes, padding = 0, seed = 0, wordnum = 0;
unsigned int LFSRval, LFSRtab[8];

/*****************************************************************/
int decode_word(word)
   char* word;

   int length, ok = 0, c, ret = 0;

   length = strlen(word);
   for (i=0; i<length; i++) {
          c = word[i];
          if ( (ok == 0) && (islower(c)) )
                 c = toupper(c);

          c -= 65;
          if ( ((c >= 0) && (c < 26)) || ((c >= 32) && (c < 58)) ) {
                 ok = -1;
                 ret += c;
          }
   }
   return ok? ret : BADWORD;
```

```
/*************************************************************************/
void InitLFSR()
{
   int i;

   /*
    * make the table mapping 8-bit uniform rv's to a binary rv with prob.
    * of outputing a 0 given by padding / 8  (for padding = 0 to 7)
    */

   for (i = 0; i < 8; LFSRtab[i++] = 1) ;
   for (i = 0; i < padding; LFSRtab[i++] = 0) ;

   /*  put initial value in shift register -- use seed, if supplied  */

   LFSRval = 0x9734;     /*  &b1001 0111 0011 0100 */
   if ((seed > 0) && (seed != LFSRval))
      LFSRval = LFSRval ^ seed;

}
/*************************************************************************/
unsigned int LFSRGetBit()
{
   unsigned int i,v,b1,b2,b3;

   for (i = 0; i < 3; i++) {   /* cycle shift reg 3 times to get 3-bit val */

      /*  new input to LFSR = b14 XOR b9 XOR b3  */

      b1 = (LFSRval >> 14) & 1;
      b2 = (LFSRval >> 9) & 1;
      b3 = (LFSRval >> 3) & 1;

          v = b1 ^ b2 ^ b3;
      LFSRval = (LFSRval << 1) | v;
   }
   v = LFSRval & 7;              /* take lower 3 bits of shift reg output */

   return LFSRtab[v];   /* map 3 bits to binary r.v value */
}
/*************************************************************************/
void get_args(argc,argv)
   char* argv[];
   int argc;
{
   if (argc < 2) {
          puts("USAGE: debonzo bonzofile [options] [-o outputfile]");
          exit(-1);
   }

   for (i = 1; i < argc; i++) {
      if (STRICMP(argv[i],"-b") == 0) {
         wordbits = atoi(argv[++i]);
         if ((wordbits < 1) || (wordbits > 8)) {
            wordbits = DEFAULTWORDBITS;
            printf("Weird wordbits value.  Wordbits = %d assumed\n",
            wordbits);
         }
         xb = 1;

      } else if (STRICMP(argv[i],"-p") == 0) {
          padding = atoi(argv[++i]);
          if ((padding < 0) || (padding > 7)) {
             padding = 0;
             printf("Weird padding value.  padding = 0 assumed\n");
          }
          xpad = 1;
```

```
    } else if (STRICMP(argv[i],"-s") == 0) {
        seed = atoi(argv[++i]);
        xseed = 1;

    } else if (STRICMP(argv[i],"-xlen") == 0) {
        i++;
        nbytes = atoi(argv[i]);
        xlen = 1;

    } else if (STRICMP(argv[i],"-o") == 0) {
        outf = 1;
        if (!(outfile = fopen(argv[++i],"wb"))) {
           printf("Can't open output file: %s\n",argv[i]);
           exit(-1);
        }

    } else if (STRICMP(argv[i],"-begin") == 0) {
        i++;
        beginsentence = atoi(argv[i]);

    } else if (STRICMP(argv[i],"-debug") == 0) {
        debug = 1;

    } else {
        if (!(infile = fopen(argv[i],"rt"))) {
           printf("Can't open input file: %s\n",argv[i]);
           exit(-1);
        }
    }
  }
  if (!infile) {
     puts("\nNo input file specified!\n");
     puts("USAGE: debonzo bonzofile [options] [-o outputfile]");
     exit(-1);
  }

}
/*****************************************************************************/
main(argc,argv)
  char* argv[];
  int argc;
{
  int i, done, bytesdone = 0, cbits = 0, wordcode, outch, bitsdone = 0;
  int INandmask, OUTandmask = (1 << OUTBITS) - 1;
  unsigned long shiftreg = 0;

  get_args(argc,argv);

  fprintf(stderr,"\n");

  while ((fscanf(infile,"%s",word) != EOF) &&
         (STRNICMP(word,triggerword,strlen(triggerword)) != 0)) ;

  if (feof(infile)) {
        printf("Couldn't find trigger word: %s\n",triggerword);
        exit(-1);
  }

  for (i = 0; i <= beginsentence; fgets(dummystring,80,infile), i++) ;

  if (feof(infile)) {
        printf("No bonzotext found %d lines after line with 'Dear'\n",
                beginsentence);
        exit(-1);
  }
```

```
if (!xb) {
   for (done = 0; !done; ) {
      fscanf(infile,"%s",word);
      if ((wordcode = decode_word(word)) != BADWORD) {
         wordbits = wordcode & 7;
         if (wordbits == 0) {
            wordbits = DEFAULTWORDBITS;
            fprintf(stderr,"Weird wordbits.  Wordbits = %d assumed\n",wordbits);
         }
         done++;
      }
   }
}

if (!xpad) {
   for (done = 0; !done; ) {
      fscanf(infile,"%s",word);
      if ((wordcode = decode_word(word)) != BADWORD) {
         padding = wordcode & 7;
         done++;
      }
   }
}


if (!xseed) {
   for (done = 0; !done; ) {
      fscanf(infile,"%s",word);
      if ((wordcode = decode_word(word)) != BADWORD) {
         seed = wordcode & 7;
         done++;
      }
   }
}


if (debug)
   fprintf(stderr,"wordbits=%d,padding=%d,seed=%d",wordbits,padding,seed);

/* and now the body begins */

InitLFSR();
INandmask  = (1 << wordbits) - 1;

if (!xlen) {                    /* find numbytes */
   while (cbits < LENBITS) {
      fscanf(infile,"%s",word);
      if ((wordcode = decode_word(word)) != BADWORD) {
         if (LFSRGetBit()) {
            wordcode ^= LFSRval;
            shiftreg = (shiftreg << wordbits) | (wordcode & INandmask);
            cbits += wordbits;
         }
         wordnum++;
      }
   }

   nbytes = shiftreg >> (cbits - LENBITS);
   shiftreg &= (1 << (cbits - LENBITS)) - 1;    /* mask out upper LENBITS */
   shiftreg = shiftreg << wordbits;
   cbits -= LENBITS;
}


if (debug)
   fprintf(stderr,"   nbytes = %d\n\n",nbytes);
```

```
  while ((fscanf(infile,"%s",word) != EOF) && (bytesdone < nbytes))
     if ((wordcode = decode_word(word)) != BADWORD) {
        if (LFSRGetBit()) {
           wordcode ^= LFSRval;
           shiftreg |= (wordcode & INandmask);
           cbits += wordbits;

           if (debug)
              fprintf(stderr,"%d: %d\t%s\t",wordnum,wordcode&INandmask,word);

           if (cbits >= OUTBITS) {
              outch = (shiftreg >> (cbits - OUTBITS)) & OUTandmask;
              shiftreg &= ((1 << (cbits - OUTBITS)) - 1);
              cbits -= OUTBITS;
              if (debug)
                 fprintf(stderr,"char = %d %c",(int) outch, (char) outch);
              else
                 putc(outch,stdout);

              if (outf)
                 fputc(outch,outfile);

              bytesdone++;
           }
           shiftreg = shiftreg << wordbits;

           if (debug)
              fprintf(stderr,"\n");

        }
        else if (debug) {
           fprintf(stderr,"%d:\t%s -- dummy \n",wordnum,word);
        }

        wordnum++;
     }

     if ((feof(infile)) && (bytesdone < nbytes))
        fprintf(stderr,"\nPremature end of input file!\n");

     fclose(infile);

     if (outf)
        fclose(outfile);

     fflush(stdout);

     fprintf(stderr,"\n");

     fprintf(stderr,"\n");

}

/*****************************************************************************/
```

A.4 BONZIFY.BAS

A.4.1 A Few Words About PowerBasic

The PowerBasic language is not far removed from other procedural languages such as C, and it is often easier to use for certain programming tasks. We anticipate that readers familiar with C or Pascal will be able to make some sense of the BONZIFY.BAS source code, particularly if the following tips are noted:

- In PowerBasic, the data type of a variable is usually given by the last character of its name (the "extension" character). Variable types used in the bonzifier source code are shown below.

`name`	(no extension)	single-precision floating-point number
`name%`		16-bit, signed integer
`name$`		character string of arbitrary length (as memory permits)
`name?`		8-bit, unsigned byte
`name??`		16-bit, unsigned integer
`name???`		32-bit, unsigned integer

- If the data-type extension is the first character of the name, then the name represents a global constant (e.g., `%MaxThings = 256`).

- All characters following a single quote on a line are considered a comment. e.g.,

```
y = SQR(-1)    ' I should fix this later
```

A.4.2 The Source Code for BONZIFY.BAS

```
usage:
BONZIFY [-i infile] [-l langfile] [-o outfile] [ -b wordbits] [-p padding]
        [-s seed] [-debug] [-check thing]


'-------------------------------------------------------------------------
'lobal constants, directives:
'-------------------------------------------------------------------------
'stack 10000
'ERROR Stack+, Bounds+, Numeric-, Overflow-

'efaultinfile$    = "INBONZO.TXT"
'efaultlangfile$ = "POETRY1.L"
'efaultoutfile$  = "BONZO1.B"
'ebugfile$        = "BONZO.LOG"
'ebug%    = 0
```

```
'-------------------------------------------------------------------------
Constants for Language generator:
'-------------------------------------------------------------------------
%RandReps     = -1
%Thing        = 0
%Literal      = 1
%Punctuation  = 2
%Probfactor   = 3
%Unknown      = 999
%LineWrap     = -1
%OutFile      = 1
%OutConsole   = 2

%MAXITS       = 10       ' how recursive we will let it be
%MaxThings    = 256
%MaxRules     = 5000
%MaxInSize= 4092 ' in bytes
        ' we must have %MaxInSize * wordbits% / 8 < 32K

'-------------------------------------------------------------------------
Constants for Encoder:
'-------------------------------------------------------------------------
wordbits% = 5
padding% = 0
check% = 0
xlen%  = 0        ' default: embed length in bonzotext
%LenBits  = 12    ' num bits we devote to the length of bonzotext payload

screen 0
color 9,0
on error goto ErrorHandler
randomize timer

call GetCommandLineArgs(infile$,langfile$,outfile$)
call LoadInFile
call GenCodePos

wordmask% = (2 ^ wordbits%) - 1
type node
   lpoint as byte
   tipe   as byte
   nval   as byte
   st     as byte
   nchars as byte
end type

dim Lpoint%(%MaxThings),Lnopts%(%MaxThings),Lshow%(%MaxThings)
dim Lonetime%(%MaxThings),Lvar$(%MaxThings),Lthing$(%MaxThings)

dim Rule$(%MaxRules),Rused%(%MaxRules),Rnodest%(%MaxRules),Rnnodes%(%MaxRules)
dim Rprobfactor%(%MaxRules)
dim huge rnode(22000) as node
dim thingstack$(%MAXITS)

for i% = 0 to %MaxThings-1
   Lvar$(i%) = ""
next i%
for i% = 0 to %MAXITS-1
   thingstack$(i%) = ""
next i%
call LearnLang
  "Lnthings:";Lnthings%,"Nrules:";nrules%;
rigtime = timer
call ProcessLang
  ,"Nrnodes:";nrnodes%
```

```
open outfile$ for output as #3
if debug% then open debugfile$ for output as #2
if check% then call CheckThing
StartOfSentence% = -1
wordnum% = inbufstart%

call FindThing("Prelude",i%,er%)
call MakeThing(i%,0,0,fail%,Op1$)       ' don't encode, not desperate

call FindThing("Body",i%,er%)
call MakeThing(i%,-1,0,fail%,Op2$)       ' not desperate (initially)

call FindThing("Ending",i%,er%)
call MakeThing(i%,0,0,fail%,Op3$)       ' don't encode, not desperate


StartOfSentence% = -1
call LWPrint(Op1$,0,%OutFile)
call LWPrint(Op2$,1,%OutFile)
call LWPrint(Op3$,0,%OutFile)

? "Words in bonzoblock:";wordnum%
? "Total Errors:";TotalErrs%
tm = timer - origtime
tmin% = tm \ 60
tsec% = tm - tmin%*60
tsec$ = str$(tsec%) : tsec$ = right$(tsec$,len(tsec$)-1)
if len(tsec$) = 1 then tsec$ = "0" + tsec$
? "Total Time: "; : print using "####"; tmin%; : ? ":";tsec$;

if xb% or xpad% or xseed% or xlen% then
   ?
   color 15 : ? "The debonzifier will need to know the following: "
end if
if xb% then
   color 13,0 : ? "   wordbits:";
   color 14,0 : ? wordbits%;
   color 15,0 : locate ,19 : ? "-b";wordbits%
end if
if xpad% then
   color 13,0 : ? "   padding:";
   color 14,0 : ? padding%;
   color 15,0 : locate ,19 : ? "-p";padding%
end if
if xseed% then
   color 13,0 : ? "   seed:";
   color 14,0 : ? seed%;
   color 15,0 : locate ,19 : ? "-s";seed%
end if
if xlen% then
   color 13,0 : ? "   length:";
   color 14,0 : ? inbytes%;
   color 15,0 : locate ,19 : ? "-xlen";inbytes%
end if

if debug% then close 2
close 3


end
      ' must have END since there is ErrorHandler code later
'-----------------------------------------------------------------------
sub ProcessLang
   shared Lpoint%(),Lnopts%(),Lthing$(),Lonetime%(),Lnthings%,Lshow%()
   shared Rule$(),Rnodest%(),Rnnodes%(),rnode(),nrnodes%,Rprobfactor%()
   shared nrules%
```

```
'? : ? "Processing Language"
color 11,0
nrnodes% = 0

for thing% = 0 to Lnthings%-1
   r% = Lpoint%(thing%)
   ting$ = Lthing$(thing%)
   for rn% = 0 to Lnopts%(thing%)-1
      call IncTimer
      rool$ = Rule$(r%)
      newrool$ = ""
      roolnodes% = 0
      cptr% = 1
      Rnodest%(r%) = nrnodes%
      while rool$ <> ""
         call GetToken(rool$,t$,tipe%)

         if tipe% = %thing then
            t1$ = t$
            if left$(t$,1) = "#" then t1$ = right$(t$,len(t$)-1)
            call FindThing(t1$,index%,er%)
            if er% then
               color 14,0 : ? "thing:"; : color 15,0 : ? ting$;
               color 14,0 : ? "  rule:"; : color 15,0 : ? Rule$(r%)
               end
            end if

            if newrool$ <> "" then newrool$ = newrool$ + " " : incr cptr%
            rnode(nrnodes%).tipe   = %thing
            rnode(nrnodes%).lpoint = index%
            rnode(nrnodes%).st     = cptr%
            rnode(nrnodes%).nchars = len(t$)

            incr nrnodes%
            incr roolnodes%
            newrool$ = newrool$ + t$
            cptr% = cptr% + len(t$)

         elseif tipe% = %Punctuation then
            t$ = mid$(t$,2,len(t$)-2)    ' get rid of quote marks
            if newrool$ <> "" then newrool$ = newrool$ + " " : incr cptr%
            rnode(nrnodes%).tipe   = %Punctuation
            rnode(nrnodes%).st     = cptr%
            rnode(nrnodes%).nchars = len(t$)

            incr nrnodes%
            incr roolnodes%
            newrool$ = newrool$ + t$
            cptr% = cptr% + len(t$)

         elseif tipe% = %ProbFactor then
            Rprobfactor%(r%) = val(t$)

         elseif tipe% = %Literal then
            t$ = mid$(t$,2,len(t$)-2)    ' get rid of quote marks

            if ting$="Prelude" or ting$="Ending" then
               if newrool$ <> "" then newrool$ = newrool$ + " " : incr cptr%
               rnode(nrnodes%).tipe   = %Literal
               rnode(nrnodes%).st     = cptr%
               rnode(nrnodes%).nchars = len(t$)

               incr nrnodes%
               incr roolnodes%
               newrool$ = newrool$ + t$
               cptr% = cptr% + len(t$)
```

```
         else
            do
               if newrool$ <> "" then newrool$=newrool$ + " " : incr cptr%

               call GetToken(t$,word$,er%)
               call EncodeWord(word$,w%)

               rnode(nrnodes%).tipe   = %Literal
               rnode(nrnodes%).nval   = w%
               rnode(nrnodes%).st     = cptr%
               rnode(nrnodes%).nchars = len(word$)

               incr nrnodes%
               incr roolnodes%
               newrool$ = newrool$ + word$
               cptr% = cptr% + len(word$)

            loop until t$ = ""
         end if
      end if
   wend

   Rnnodes%(r%) = roolnodes%
   Rule$(r%) = newrool$

   incr r%
next rn%
next thing%

' now check the ConsoleEcho rule

call FindThing("ConsoleEcho",i%,er%)
if er% then ? "'ConsoleEcho' not defined, so nothing will be seen" : exit sub
r% = Lpoint%(i%)
nodeptr% = Rnodest%(r%)
for j% = 0 to Rnnodes%(r%)-1
   if rnode(nodeptr%).tipe <> %thing then
      ? "Error in definition of 'ConsoleEcho'" : exit sub
   end if
   thingptr% = rnode(nodeptr%).lpoint
   Lshow%(thingptr%) = -1
   incr nodeptr%
next j%
color 15,0

end sub
-------------------------------------------------------------------------
sub LearnLang
   shared Lpoint%(),Lnopts%(),Lthing$(),Lonetime%(),Lnthings%,Rule$()
   shared langfile$,nrules%

open langfile$ for input as #2

nthings% = 0, nrules% = 0, nopts% = 0, linecont% = 0

color 9,0 : ? "   Loading language";
color 13,0
t$ = ""
while not eof(2)
   line input #2, t$
   origline$ = t$
   call KillInitSpace(t$)
   call KillComment(t$)
   call IncTimer
   if t$ <> "" then
      KillEndSpace(t$)
      if right$(t$,1) = "_" then
```

```
      a$ = a$ + left$(t$,len(t$)-1)
   else
      a$ = a$ + t$
      i% = instr(a$,"->")

      if i% > 0 then             ' possibly new thing
         call GetToken(a$,t$,tipe%)
         call GetToken(a$,b$,tipe%)
         if b$ = "->" and t$ <> "" and a$ <> "" then
            if Lnthings% > 0 then Lnopts%(Lnthings%-1) = nopts%
            nopts% = 1
            call CheckOneTime(t$,ot%)
            if ot% then Lonetime%(Lnthings%) = -1
            Lthing$(Lnthings%) = t$
            Lpoint%(Lnthings%) = nrules%

            m% = instr(a$,"|")
            while m% > 0
               q$ = left$(a$,m%-1)
               Rule$(nrules%) = q$
               a$ = right$(a$,len(a$)-m%)
               incr nopts%
               incr nrules%
               m% = instr(a$,"|")
            wend
            Rule$(nrules%) = a$
            incr Lnthings%
            incr nrules%
         else
            color 14,0 : ? "Input format error: ";
            color 15,0 : ? origline$
            color 14,0 : ? "Line Ignored"
         end if

      elseif Lnthings% = 0 then      ' first line not a definition
         color 14,0 : ? "Error: first line should be a Thing definition"
         end
      else
         m% = instr(a$,"|")
         while m% > 0
            q$ = left$(a$,m%-1)
            Rule$(nrules%) = q$
            Rule$(nrules%) = left$(a$,m%-1)
            a$ = right$(a$,len(a$)-m%)
            incr nopts%
            incr nrules%
            m% = instr(a$,"|")
         wend
         Rule$(nrules%) = a$
         incr nopts%
         incr nrules%
      end if
      a$ = ""
   end if
end if
wend
Lnopts%(Lnthings%-1) = nopts%
close 2
'? : color 14,0 : ? string$(70,"-")
color 15,0
end sub
'-------------------------------------------------------------------------
sub MakeThing(tm%,encode%,desperate%,fails%,pout$)
   shared Lpoint%(),Lnopts%(),Lshow%(),Rule$(),Rused%(),level%,debug%
   shared Lthing$(),Lonetime%(),Lvar$(),Lnthings%,inpos%,inlen%,inbuf%()
   shared wordnum%,Rnodest%(),Rnnodes%(),RProbFactor%(),rnode(),TotalErrs%
   shared thingstack$(),wordmask%
```

```
local r$,t$,Op$,newOp$,i%,k%,nreps%,failed%,tomake$,tfails%,t1%,t2%
local rulesleft%,rtried%(),nfailures%(),rpf%(),minfails%,rulenum%,failed%
local oldinpos%,rnodeptr%(),rout$(),rleft$(),xinpos%()
local rrep%,rnptr%,tipe%,rnodeend%(),rnend%

tomake$ = Lthing$(tm%)
i% = tm%

'thingstack$(level%) = tomake$
'for k% = 0 to level%-1
'        if thingstack$(k%) = tomake$ then
'      ? "Whoah!  Recursive definition in language: ";
'      color 15,0 : ? tomake$; : ?
'      if debug% then print #2, "Recursive definition: ";tomake$
'      fails% = 999
'      pout$ = ""
'      exit sub
'   end if
'next k%

incr level%
if debug% then Print #2, space$(level%);level%;":";tomake$

if level% = %MAXITS then
   fails% = 999
   pout$ = ""
   decr level%
   exit sub
   if debug% then print #2, "Too recursive for me!!!"
end if
nreps% = 1
' if level% = 1 then call ClearOneTimers

rulesleft% = Lnopts%(i%)

dim rtried%(rulesleft%),nfailures%(rulesleft%),rout$(rulesleft%)
dim rpf%(rulesleft%),rnodeptr%(rulesleft%),rnodeend%(rulesleft%)
dim rinpos%(rulesleft%),rwordnum%(rulesleft%)

k% = Lpoint%(i%)
for j% = 0 to rulesleft%-1
   rout$(j%) = ""
   rnodeptr%(j%) = Rnodest%(k%)
   rnodeend%(j%) = rnodeptr%(j%) + Rnnodes%(k%)
   rinpos%(j%)   = inpos%
   rwordnum%(j%) = wordnum%

   rpf%(j%)      = RProbFactor%(k%)
   nfailures%(j%) = 0
   incr k%
next j%

newrule% = -1
done% = 0
fails% = 0
while not done%
   call IncTimer
   k$ = inkey$
   if k$ = chr$(27) then close #3 : end
   if k$ = "d" then debug% = 1 - debug%  : ? "debug: ";debug%
   if newrule% then
      call GetRuleNum(rtried%(),nfailures%(),rpf%(),rulesleft%,rulenum%)
      if encode% then
         rtried%(rulenum%) = -1
         wordnum% = rwordnum%(rulenum%)
         inpos% = rinpos%(rulenum%)
```

```
      Op$ = rout$(rulenum%)
   end if
   rnptr% = rnodeptr%(rulenum%)
   rnend% = rnodeend%(rulenum%)
   r$ = Rule$(Lpoint%(i%) + rulenum%)
   newrule% = 0
   if debug% then Print #2, space$(level%);level%;" rule :";r$
end if
failed% = 0 ' applies to current node only; rule tally kept in nfailures%()

   ' consider the next token in the rule

t$ = mid$(r$,rnode(rnptr%).st,rnode(rnptr%).nchars)
tipe% = rnode(rnptr%).tipe

if tipe% = %thing and left$(t$,1) = "#" then
   if inpos% < inlen% then rrep% = -1
else
   rrep% = 0
end if

if tipe% = %Punctuation then
   newOp$ = t$

elseif tipe% = %Literal then
   newOp$ = t$
   if encode% and (inpos% < inlen%) then
      if wordnum% < 0   then   ' preamble   use b=3
            t1% = inbuf%(wordnum%)
         incr wordnum%
         t2% = rnode(rnptr%).nval AND 7
         if t1% <> t2% then incr failed%
      else
         call CheckForEncode(wordnum%,e%,cv%)
         incr wordnum%
         if e% then
            t1% = inbuf%(inpos%)
            incr inpos%
            t2% = (rnode(rnptr%).nval XOR cv%) AND wordmask%
            if t1% <> t2% then incr failed%
         end if
      end if
   end if
elseif tipe% = %thing then
   k% = rnode(rnptr%).lpoint
   call MakeThing(k%,encode%,0,failed%,newOp$)

end if

call OutputAppend(Op$,newOp$)
incr rnptr%

if encode% then
   nfailures%(rulenum%)=nfailures%(rulenum%) + failed%
   rnodeptr%(rulenum%) = rnptr%
   rinpos%(rulenum%) = inpos%
   rwordnum%(rulenum%) = wordnum%
   rout$(rulenum%) = Op$
   rfails% = nfailures%(rulenum%)

   minfails% = 32700
   for j% = 0 to rulesleft%-1
      if nfailures%(j%)<minfails% then minfails%=nfailures%(j%)
   next j%

   if rnptr% = rnend% and rfails% <= minfails% then
      done% = -1       ' current rule is done and as good as the rest
```

```
      elseif rnptr% = rnend% or rfails% > minfails% then
         newrule% = -1    ' a better rule exists
         j% = 0
         while not done% and j% < rulesleft%
            if nfailures%(j%)=minfails% and rnodeptr%(j%)=rnodeend%(j%) then
               done% = -1
               Op$ = rout$(j%)
               rfails% = minfails%
               inpos% = rinpos%(j%)
               wordnum% = rwordnum%(j%)
            end if
            incr j%
         wend
      end if

      if done% then
         tfails% = tfails% + rfails%
         if Lthing$(i%) = "Body" then TotalErrs% = tfails%

         if debug% then
            Print #2, space$(level%);level%;" prog out: ";Op$;
            Print #2,"   Errors: ";tfails%
         end if

         if rrep% and inpos% < inlen% then
            decr rnptr%
            nfailures%(rulenum%) = 0
            done% = 0
         end if
      end if

   else       ' NO ENCODE
      if rnptr% = rnend% then done% = -1
   end if
vend
if Lshow%(i%) then call LWPrint(Op$,encode%,%OutConsole)

rase rtried%,nfailures%,rout$,rnodeptr%,rnodeend%,rinpos%

   ' check for a one-time variable assignment here

ails% = tfails%
out$ = Op$
ecr level%
nd sub
----------------------------------------------------------------------------
ub FindThing(tomake$,i%,er%)
   shared Lthing$(),Lnthings%
r% = 0
% = 0
hile (Lthing$(i%) <> tomake$) and i% < Lnthings%
   incr i%
end

f i% = Lnthings% then
   color 13,0 : ? : ? tomake$;
   color 14,0 : ? " not defined in Language"
   er% = -1
nd if

nd sub
----------------------------------------------------------------------------
ub GetToken(cc$,tt$,tipe%)
   local c$,t$

all KillInitSpace(cc$)
```

```
c$ = cc$
t$ = ""
if c$ <> "" then

   ' check for punctuation/formatting only!!!

   if left$(c$,1) = chr$(34) then          ' the double quote
      i% = instr(right$(c$,len(c$)-1),chr$(34))
      if i% = 0 then tipe% = %Unknown : exit sub
      if i% = 1 then       ' two quotes in a row = "
         t$ = string$(3,chr$(34))
      else
         t$ = left$(c$,i%+1)
      end if
      c$ = right$(c$,len(c$)-i%-1)
      tipe% = %Literal

      punc% = -1
      i% = 2
      while punc% and i% < len(t$)
         if i% <= len(t$)-2 then
            x$ = mid$(t$,i%,2)
            if ucase$(x$) = "\N" then
               i% = i% + 2
            else
               x% = asc(mid$(t$,i%,1)) - 65
               if (x%>=0 AND x%<26) OR (x%>31 AND x%<58) then punc%=0
               incr i%
            end if
         else
            x% = asc(mid$(t$,i%,1)) - 65
            if (x%>=0 AND x%<26) OR (x%>31 AND x%<58) then punc%=0
            incr i%
         end if
      wend
      if punc% then tipe% = %Punctuation

   elseif left$(c$,1) = "<" then          ' random reps
      c$ = right$(c$,len(c$)-1)
      call GetToken(c$,t$,t%)

      tipe% = %RandReps

   elseif left$(c$,1) = "[" then          ' probability factor
      c$ = right$(c$,len(c$)-1)
      i% = instr(c$,"]")
      if i% = 0 then tipe% = %Unknown : cc$ = "" : tt$ = "" : exit sub
      t$ = left$(c$,i%-1)
      c$ = right$(c$,len(c$)-i%)

      tipe% = %ProbFactor

   else
      i% = instr(c$," ")
      t% = instr(c$,chr$(9))
      if t% > 0 and (i% = 0 or t% < i%) then i% = t%

      if i% > 0 then
         t$ = left$(c$,i%-1)
         c$ = right$(c$,len(c$)-i%)
      else
         t$ = c$
         c$ = ""
      end if
      tipe% = %Thing
   end if
ıd if
```

```
call KillInitSpace(c$)
call KillInitSpace(t$)
cc$ = c$
tt$ = t$
end sub
'-------------------------------------------------------------------------
sub KillInitSpace(a$)

while a$ <> "" and ((left$(a$,1) = " ") or (left$(a$,1) = chr$(9)))
   a$ = right$(a$,len(a$)-1)
wend

end sub
'-------------------------------------------------------------------------
sub KillEndSpace(a$)

while a$ <> "" and ((right$(a$,1) = " ") or (right$(a$,1) = chr$(9)))
   a$ = left$(a$,len(a$)-1)
wend

end sub
'-------------------------------------------------------------------------
sub KillComment(a$)

i% = instr(a$,"//")
if i% > 0 then a$ = left$(a$,i%-1)

end sub
'-------------------------------------------------------------------------
sub usage

  "Usage:"

  "bonzify [-i infile] [-l langfile] [-o outfile] [-b wordbits]"
  "        [-p padding] [-s seed] [-check thing] [-debug]"

nd sub
'-------------------------------------------------------------------------
ub GetCommandLineArgs(infile$,langfile$,outfile$)
  shared defaultinfile$,defaultlangfile$,defaultoutfile$,debug%,wordbits%
  shared padding%,seed%,check%,xlen%,xb%,xpad%,xseed%,tocheck$

im dynamic args$(16)
args% = 0

$ = command$
infile$   = defaultinfile$
langfile$ = defaultlangfile$
outfile$  = ""

ile c$ <> ""
  call GetToken(c$,args$(nargs%),tipe%)
  if args$(nargs%) <> "" then incr nargs%
nd

 = 0
ile i% < nargs%
  args$(i%) = lcase$(args$(i%))
  if args$(i%) = "-b" then
     incr i%
     wordbits% = val(args$(i%))
     if wordbits% < 1 or wordbits% > 8 then usage : end

  elseif args$(i%) = "-p" then
     incr i%
     padding% = val(args$(i%))
     if padding% < 0 or padding% > 8 then usage : end
```

```
   elseif args$(i%) = "-s" then
      incr i%
      seed% = val(args$(i%))

   elseif args$(i%) = "-debug" then
      debug% = 1

   elseif args$(i%) = "-check" then
      incr i%
      tocheck$ = args$(i%)
      check% = -1

   elseif args$(i%) = "-i" then
      incr i%
      cinfile$ = args$(i%)

   elseif args$(i%) = "-l" then
      incr i%
      clangfile$ = args$(i%)

   elseif args$(i%) = "-xl" then
      xlen% = -1

   elseif args$(i%) = "-xb" then
      xb% = -1

   elseif args$(i%) = "-xp" then
      xpad% = -1

   elseif args$(i%) = "-xs" then
      xseed% = -1

   elseif args$(i%) = "-o" then
      incr i%
      coutfile$ = args$(i%)
   end if
   incr i%
wend

' try to open input file

exists% = 0
while not exists%
   call FileExists(cinfile$,exists%)
   if exists% then
      infile$ = cinfile$
   else
      color 13,0 : ? "Input file: ";cinfile$;" not found."
      call getparm("Input file name ["+defaultinfile$+"]",cinfile$)
      if cinfile$ = "" then cinfile$ = defaultinfile$
   end if
wend
'color 14,0 : ? "Infile: ";
'color 15,0 : ? infile$


' try to open language file

exists% = 0
while not exists%
   call FileExists(clangfile$,exists%)
   if exists% then
      langfile$ = clangfile$
   else
      color 13,0 : ? "Language file: ";clangfile$;" not found."
      call getparm("Language file name ["+defaultlangfile$+"]",clangfile$)
```

```
      if clangfile$ = "" then clangfile$ = defaultlangfile$
   end if
wend
'color 14,0 : ? "Language file: ";
'color 15,0 : ? langfile$


if coutfile$ = "" then
   call getparm("Output file name ["+defaultoutfile$+"]",outfile$)
   if outfile$ = "" then outfile$ = defaultoutfile$
else
   outfile$ = coutfile$
end if

erase args$
end sub
'-------------------------------------------------------------------------------
sub FileExists(filename$,exists%)
   shared filenotfound%

filenotfound% = 0
open filename$ for input as #5
if filenotfound% then
   exists% = 0
   filenotfound% = 0
else
   exists% = -1
   close 5
end if

end sub
'-------------------------------------------------------------------------------
sub getparm(q$,a$)

color 11
?q$;"? ";
color 15
input "",a$
color 14

end sub
'-------------------------------------------------------------------------------
sub CheckOneTime(t$,ot%)

ot% = 0
i% = instr(t$,"*")
if i% > 0 then
   t$ = left$(t$,i%-1)
   ot% = -1
end if
end sub
'-------------------------------------------------------------------------------
sub CheckforPunctuation(t$,isap%)
   static firsttime%,p$(),np%

if firsttime% = 0 then
   firsttime% = -1
   np% = 10
   dim p$(np%)
   p$(0) = "." : p$(1) = ";" : p$(2) = "," : p$(3) = "!" : p$(4) = ":"
   p$(5) = "?" : p$(6) = "'" : p$(7) = chr$(34) : p$(8) = "-"
   p$(9) = chr$(13)
end if

isap% = 0
t$ = left$(t$,1)
for i% = 0 to np%-1
```

```
   if 1$ = p$(i%) then isap% = -1
next i%

end sub
'-------------------------------------------------------------------------
sub IncTimer
   static t%,tcptr%, firsttime%,tchar$(),lasttime
        shared inpos%,inlen%,origtime,debug%,TotalErrs%

if firsttime% = 0 then
   firsttime% = 1
   dim tchars$(8)
   tchar$(0) = "|" : tchar$(1) = "/" : tchar$(2) = "-"
   tchar$(3) = "\" : tchar$(4) = "|" : tchar$(5) = "/"
   tchar$(6) = "-" : tchar$(7) = "\"
   tcptr% = 0
      t% = 0
   lasttime = -1
end if
y% = Csrlin
x% = Pos(0)

if t% MOD 17 = 0 then
   locate 1,78
   ? tchar$(tcptr%);
   incr tcptr%
   if tcptr% = 8 then tcptr% = 0
end if
if inpos% > 0 and t% MOD 20 = 0 then
   locate 1,62 : ? space$(18);
   locate 2,62 : ? space$(18);
   locate 1,66 : ? inpos%;"of";inlen%;
   locate 2,62 : ? "Errs:";TotalErrs%;

   t% = 0
   tm = timer - origtime
   if tm <> lasttime then
      tmin% = tm \ 60
      tsec% = tm - tmin%*60
      tsec$ = str$(tsec%) : tsec$ = right$(tsec$,len(tsec$)-1)
      if len(tsec$) = 1 then tsec$ = "0" + tsec$
         locate 2,74 : print using "####"; tmin%; : ? ":";tsec$;
      end if
   lasttime = tm
nd if
ncr t%
ocate y%,x%

nd sub
-------------------------------------------------------------------------
ub LWprint(Op$,changecolor%,dst%)
  shared outfile$,inpos%,StartOfSentence%

olor 15,0

p$ = Op$
% = instr(tp$,"\n")
hile i% > 0
  tp$ = left$(tp$,i%-1)+chr$(13)+chr$(10)+right$(tp$,len(tp$)-i%-1)
  i% = instr(tp$,"\n")
end

% = 1
tp$ = tp$
p$ = ""
or i% = 1 to len(otp$)
  c$ = mid$(otp$,i%,1)
```

```
  if c$ = "." or c$ = "!" or c$ = "?" or c$ = ":" then
     StartOfSentence% = -1
     tp$ = tp$ + c$
  elseif StartOfSentence% and (asc(c$) < 123 and asc(c$) > 64) then
     StartOfSentence% = 0
     tp$ = tp$ + ucase$(c$)
  else
     tp$ = tp$ + c$
  end if
ext i%

hile (instr(tp$,chr$(13)) > 0) OR (%LineWrap and (len(tp$) > 78))
  iip% = instr(tp$,chr$(13))
  if iip% > 0 and iip% <= 78 then
     if dst% = %OutFile then
        print #3, left$(tp$,iip%-1)
     else
        call cprint(changecolor%,left$(tp$,iip%-1))
        ?
     end if
     tp$ = right$(tp$,len(tp$)-iip%-1)
  else
     h% = 78
     while h% > 1 and mid$(tp$,h%,1) <> " "
        decr h%
     wend
     if dst% = %OutFile then
        print #3, left$(tp$,h%)
     else
        call cprint(changecolor%,left$(tp$,h%))
        ?
     end if
     tp$ = right$(tp$,len(tp$)-h%)
  end if
nd
f dst% = %outfile then
  print #3,tp$;
lse
  call cprint(changecolor%,tp$)
d if

d sub
--------------------------------------------------------------------------
b cprint(e%,t$)
  static lwordnum%,firsttime%
  shared inbufstart%

 firsttime% = 0 then
  lwordnum% = inbufstart%
  firsttime% = 1
d if

  e% = 0 then
  color 15,0
  ? t$;
  exit sub
 d if
 ile len(t$) > 0
  t% = 1
  while mid$(t$,t%,1) = chr$(32) or mid$(t$,t%,1) = chr$(9) and t%<len(t$)
     incr t%
  wend
  if t% = len(t$) then exit sub    ' just spaces & tabs -- nothing to print
  while mid$(t$,t%,1)<>chr$(32) and mid$(t$,t%,1)<>chr$(9) and t%<len(t$)
     incr t%
  wend
  w$ = left$(t$,t%)
```

```
   if t% < len(t$) then
      t$ = right$(t$,len(t$)-t%)
   else
      t$ = ""
   end if
   color 15,0

    punct% = -1
   for i% = 1 to len(w$)
      r$ = mid$(w$,i%,1)
      r% = asc(r$)-65
      if (r% >= 0 and r% < 26) or (r% >= 32 and r% < 58) then punct% = 0
   next i%

   if not punct% then
      if lwordnum% < 0 then
         color 14,0
      else
            call CheckForEncode(lwordnum%,c%,cv%)
         if c% then color 14,0
      end if
      incr lwordnum%
   end if
   ? w$;
wend

end sub
'-------------------------------------------------------------------------

ErrorHandler:

filenotfound%=0
if err=53 then filenotfound% = 1 : resume next
s$="program error "+str$(err)+" at "+str$(eradr)
color 15,0
? s$
end

'-------------------------------------------------------------------------
sub EncodeWord(word$,c%)
   shared wordmask%
:% = 0
firstchar% = -1
for i% = 1 to len(word$)
   r$ = mid$(word$,i%,1)
   if firstchar% then r$ = ucase$(r$)
   r% = asc(r$)-65
   if (r% >= 0 and r% < 26) or (r% >= 32 and r% < 58) then
      firstchar% = 0
      c% = c% + r%
   end if
ext i%

   ' can't do "c% = c% AND wordmask%" cuz preamble may have b=3
% = c% AND 255

nd sub
-------------------------------------------------------------------------
ub LoadInFile
   shared infile$,inbuf%(),inpos%,inlen%,inbytes%,inbufstart%
   shared wordbits%,debug%,xlen%,xb%,xpad%,xseed%,padding%,seed%


im inbuf%(-3 TO %MaxInSize * 8 / wordbits% + 1)
nbufstart% = 0

 format of full preamble:  -3  -2  -1  0
```

```
'                              b   p   s  start of body
' all preamble fields are optional

if xseed%=0 then decr inbufstart% : inbuf%(inbufstart%) = seed%

if xpad% =0 then decr inbufstart% : inbuf%(inbufstart%) = padding%

if xb%   =0 then decr inbufstart% : inbuf%(inbufstart%) = wordbits%

' zero inbuf
'color 9,0 : ? "   Loading Infile";

inbytes% = 0
if xlen% then
   inlen% = 0 : cbits% = 0
   shiftreg% = 0
else
   inlen% = %LenBits \ wordbits% : cbits% = %LenBits MOD wordbits%
   shiftreg% = 0
end if

open infile$ for binary as #1
while not eof(1)
   get$ 1,1,l$
   linepos% = 1
   while linepos% <= len(l$)
      incr inbytes%
      x% = asc(mid$(l$,linepos%,1))
      shiftreg% = shiftreg% OR x%
      cbits% = cbits% + 8

      while cbits% >= wordbits%
         inbuf%(inlen%) = shiftreg% \ 2^(cbits%-wordbits%)
         incr inlen%
         shiftreg% = shiftreg% AND (2^(cbits%-wordbits%) - 1)
         cbits% = cbits% - wordbits%
      wend
      shiftreg% = shiftreg% * 2^8
      incr linepos%
   wend
'end

 bits% = cbits%
'hile lbits% > 0
   cbits% = cbits% + 8   ' as if new value were loaded in

   while (cbits% >= wordbits%) and (lbits% > 0)
      inbuf%(inlen%) = shiftreg% \ 2^(cbits%-wordbits%)
      incr inlen%
      shiftreg% = shiftreg% AND (2^(cbits%-wordbits%) - 1)
      cbits% = cbits% - wordbits%
      lbits% = lbits% - wordbits%
   wend
   shiftreg% = shiftreg% * 2^8   ' load in dummy val
 end
 lose 1

f xlen% = 0 then
   wordmask& = 2^wordbits% - 1
   nb?? = inbytes%
   if (%LenBits MOD wordbits%) <> 0 then
      nb?? = nb?? * 2 ^ (wordbits%  - (%LenBits MOD wordbits%))
   end if

   for i% = CEIL(csng(%LenBits) / csng(wordbits%))-1 to 0 step -1
      inbuf%(i%) = inbuf%(i%) OR (nb?? AND wordmask&)
      nb?? = nb?? \ 2^wordbits%
```

```
   next i%
end if

if debug% then
   open "test.dat" for output as #1
   print #1,"inlen = ";inlen%,"nbytes = ";inbytes%
   print #1,"wordbits = ";wordbit%,"padding = ";padding%,"seed = ";seed%
   print #1,""
   for i% = inbufstart% to inlen%-1
      b$ = bin$(inbuf%(i%))
      while len(b$) < wordbits%
         b$ = "0" + b$
      wend
      print #1, i%,inbuf%(i%),b$
   next i%
   close 1
end if
inpos% = 0
end sub
'-------------------------------------------------------------------------
sub ClearOneTimers
   shared Lvar$(),Lnthings%

for k% = 0 to Lnthings%-1    ' clear the one-timers
   Lvar$(k%) = ""
next k%

end sub
'-------------------------------------------------------------------------
sub InsertTypos(tomake$,c%,t%)

' to be implemented in the future

end sub
'-------------------------------------------------------------------------
sub GetRuleNum(rtried%(1),nfailures%(1),rprobfactors%(1),nr%,rulenum%)
   static firsttime%,pdf%()

%MaxNrules = 1000
if firsttime% = 0 then
   firsttime% = -1
   dim pdf%(%MaxNrules)
end if
if nr% > %MaxNrules then beep : ? "increase MaxNrules!" : delay 1 : end

minfails% = 32000
rulesleft% = 0
s% = 0
for i% = 0 to nr%-1
   if rtried%(i%) then
      pdf%(i%) = -1
   else
      if rprobfactors%(i%)= 0 then
         s% = s% + 1
      else
         s% = s% + rprobfactors%(i%)
      end if
      pdf%(i%) = s%
      incr rulesleft%
   end if
   if nfailures%(i%) < minfails% then
      minrule% = i%
      minfails% = nfailures%(i%)
   end if
next i%

if rulesleft% = 0 then       ' pick min so far
```

```
   rulenum% = minrule%
else
   d = rnd * CSNG(s%)
   d% = fix(d)
   rulenum% = 0
   while pdf%(rulenum%) <= d% and rulenum% < nr%-1
      incr rulenum%
   wend
end if

end sub
'-------------------------------------------------------------------------
sub OutputAppend(Op$,newOp$)

if Op$ = "" then
   Op$ = newOp$
elseif right$(Op$,1) = chr$(10) or right$(Op$,2) = "\n" then
   Op$ = Op$ + newOp$
else
   call CheckforPunctuation(left$(newOp$,1),isap%)
   if isap% or right$(Op$,1) = chr$(34) then
      Op$ = Op$ + newOp$
   else
      Op$ = Op$ + " " + newOp$
   end if
end if

end sub
'-------------------------------------------------------------------------
sub CheckThing
   shared Lpoint%(),Lnopts%(),Lshow%(),Rule$(),Rused%(),level%,debug%
   shared Lthing$(),Lonetime%(),Lvar$(),Lnthings%,inpos%,inlen%,inbuf%()
   shared Rnodest%(),Rnnodes%(),rnode(),TotalErrs%,tocheck$,wordbits%

dim found%(256)
call FindThing(tocheck$,i%,er%)
if er% then end

nrules% = Lnopts%(i%)
ruleptr% = Lpoint%(i%)
? : ? "Thing: ";tocheck$
? "nrules: ";nrules%

for j% = 0 to Lnopts%(i%)-1
   nodeptr% = Rnodest%(ruleptr%)
   if rnode(nodeptr%).tipe = %Literal then
      a$ = mid$(Rule$(ruleptr%),rnode(nodeptr).st, rnode(nodeptr).nchars)
      v% = rnode(nodeptr%).nval
      incr found%(v%)
      v$ = str$(v%)
      if v% < 10 then
         v$ = "0" + right$(v$,1)
      else
         v$ = right$(v$,len(v$)-1)
      end if
      ? v$,a$
      print #3, v$,a$
   end if
   incr ruleptr%
next j%

? "9999    Missing: ";
print #3, "9999    Missing: ";
for i% = 0 to 2^wordbits%-1
   if found%(i%)=0 then ? i%; : print#3,i%;
next i%
print #3,""
```

```
close 3
end

end sub
'-----------------------------------------------------------------------
sub GenCodePos
   shared padding%,seed%,inlen%,codepos%(),codeval%(),debug%,LFSRval??

dim codepos%(inlen%),codeval%(inlen%)

call InitLFSR(padding%,seed%)

wordnum% = -1
for i% = 0 to inlen%-1
   b% = 0
   while b% = 0
      call LFSRGetBit(b%)
      incr wordnum%
   wend
   codepos%(i%) = wordnum%
   codeval%(i%) = LFSRval??
next i%

if debug% then
   ? "ncodes = ";inlen%
   ? "nwords = ";wordnum%+1
   ?
   for i% = 0 to inlen%-1
      ? codepos%(i%);": ";codeval%(i%);" ";
   next i%
   ?
end if

end sub
'-----------------------------------------------------------------------
sub InitLFSR(p%,s%)
   shared LFSRval??,LFSRtab%()

' p% is padding
' s% is seed

' make the table mapping 8-bit uniform rv's to a binary rv with probability
' of outputing a 0 given by p% / 8  (for p% = 0 to 7)

dim LFSRtab%(8)
for i% = 0 to 7
   LFSRtab%(i%) = 1
next i%
if p% > 8 then p% = 8    ' signifies "no encoding" mode
for i% = 0 to p%-1
   LFSRtab%(i%) = 0
next i%
' put initial value in shift register -- use seed, if supplied

LFSRval?? = &b1001011100110100
if s% > 0 and s% <> LFSRval?? then LFSRval?? = LFSRval?? XOR s%

end sub
'-----------------------------------------------------------------------
sub GetBit(n??,b%,o%)

o% = 0
if (n?? AND (2^b%)) <> 0 then o% = 1

end sub
'-----------------------------------------------------------------------
sub LFSRGetVal(o%)
```

```
   shared LFSRval??

for i% = 1 to 3
   ' new input to LFSR = b14 XOR b3 XOR b9

   call GetBit(LFSRval??,14,b1%)
   call GetBit(LFSRval??,3,b2%)
   call GetBit(LFSRval??,9,b3%)

   o% = (b1% XOR b2%) XOR b3%

   t& = ((LFSRval?? * 2) AND 65535) + o%
   LFSRval?? = t&
next i%
o% = LFSRval?? AND 7

end sub
'-------------------------------------------------------------------------
sub LFSRGetBit(o%)
   shared LFSRval??,LFSRtab%()

call LFSRGetVal(v%)    ' get 3-bit random value
o% = LFSRtab%(v%)

end sub
'-------------------------------------------------------------------------
sub CheckForEncode(i%,e%,cv%)
   shared codepos%(),codeval%(),inlen%,wordmask%
   static ptr%       ' initialize ptr to 0

done% = 0
while not done%
   if i% = codepos%(ptr%) then
      e% = -1 : done% = -1
        cv% = codeval%(ptr%) AND wordmask%

   elseif i% < codepos%(ptr%) then
      if ptr% = 0 then
         e% = 0 : done% = -1
      elseif i% > codepos%(ptr%-1) then
         e% = 0 : done% = -1
      else
         decr ptr%
      end if

   else    ' i% > codepos%(ptr%)
      if ptr% = inlen%-1 then
         e% = 0 : done% = -1
      elseif i% < codepos%(ptr%+1) then
            e% = 0 : done% = -1
      else
         incr ptr%
      end if
   end if
wend

end sub
'-------------------------------------------------------------------------
```

APPENDIX B: SAMPLE LANGUAGE FILE

The language file given below is used to generate bonzotext containing lists of names for grunge bands. It was used to produce the example bonzotext given in Fig. 3.2. The format of language files used by the bonzifier is described in Section 3.5.

```
// "GRUNGE.L"
// used to generate list of (fictitious) grunge band names

ConsoleEcho -> Prelude BandName Ending

Prelude ->    "Dear William: \n" _
       "    I'm sorry I haven't written you sooner. I've been trying to come\n" _
       "up with a name for my grunge band.  Here are my best ideas so far: \n \n"

Ending ->     "\n    Tell me if you have any favorites.\n" _
              "              Adam \n \n"

Body ->       #BandName

BandName ->   "" Name "" "\n"

Name ->       FrontMan "and the" PNoun        [30]
              FrontMan "&" PNP
              SNP | PNP
              SNoun SNoun [2] | SNP SNoun
              Adj
              TV-Adj PNP
              Number Adj PNoun

FrontMan ->   PersonName  | PersonName PersonName | Adj PersonName
              PersonName SNoun | SNoun

SNP -> SDet SNoun       [3]             // Singular Noun Phrase
       SDet Adj SNoun
       Adj SNoun
       SNoun

PNP -> PDet PNoun       [5]            // Plural Noun Phrase
       PDet Adj PNoun   [3]
       PDet SNoun PNoun [2]
       SNoun PNoun      [2]
       Adj PNoun
       Number PNoun
       PNoun

// Determiners to modify singular nouns

SDet ->       "the" [20] | "a" [5]
              "another"  | "some" | "this"   | "that"

// Determiners to modify plural nouns

PDet ->       "the" [20] | "those" | "these" | "some"

// Determiners indicating possession

PosDet ->     "my"  | "your" | "our" | "his" | "her" | "thy"
```

```
Number ->     "fourteen"    | "eighteen"    | "sixty-nine"    | "eighty-one"
              "eight"       | "seventy-five"| "nineteen"      | "thirty-six"
              "fifty-two"   | "forty-nine"  | "twenty-six"    | "two"
              "four"        | "eleven"      | "thirty-seven"  | "thirteen"

// Singular nouns

SNoun ->      "ghost"       | "weed"        | "bellboy"       | "lover"
              "sociologist" | "king"        | "minstrel"      | "apocalypse"
              "mold"        | "haircut"     | "curd"          | "wagon"
              "crust"       | "bastard"     | "aerosol"       | "boogie"
              "funk"        | "coroner"     | "mistress"      | "cod"
              "scum"        | "patriarch"   | "critter"       | "slug"
              "spud"        | "puppy"       | "ambush"        | "madam"
              "rubber"      | "killer"      | "sludge"        | "muffin"
              "raisin"      | "death"       | "vitriol"       | "mud"
              "art"         | "replacement" | "cow"           | "bus"
              "absorber"    | "jalopy"      | "judge"         | "fuzz"
              "obsession"   | "wax"         | "torture"       | "cadaver"
              "voodoo"      | "mascara"     | "fungus"        | "tonsil"

// Plural nouns

PNoun ->      "rabbits"     | "brothers"    | "eggs"          | "monks"
              "mushrooms"   | "delegates"   | "fish"          | "dragons"
              "flag-wavers" | "dogs"        | "cramps"        | "golfers"
              "punks"       | "breasts"     | "grocers"       | "carcasses"
              "bodies"      | "thieves"     | "lizards"       | "journalists"
              "angels"      | "muscles"     | "censors"       | "doctors"
              "victims"     | "boys"        | "eyelashes"     | "flowers"
              "cufflinks"   | "clergymen"   | "daughters"     | "puppies"
              "heads"       | "devils"      | "fjords"        | "musicians"
              "enzymes"     | "probes"      | "sisters"       | "growths"
              "bears"       | "coeds"       | "torpedoes"     | "barbarians"
              "beggars"     | "lovers"      | "stones"        | "kings"
              "breakfasts"  | "shoes"       | "skeptics"      | "paperbacks"

PersonName -> "Kenny"       | "Harry"       | "Eddy"          | "Sara"
              "Lola"        | "Herbert"     | "Sheldon"       | "Bunny"
              "Albert"      | "Bonnie"      | "Ginger"        | "Eli"
              "Mickey"      | "Jonnie"      | "Reginald"      | "Howard"
              "Alexis"      | "Judas"       | "Drake"         | "David"
              "Robert"      | "Rosy"        | "Troy"          | "James"
              "Jimbo"       | "Wolfgang"    | "Deborah"       | "Adam"

/ Adjectives which may also be used as transitive verbs

V-Adj ->      "rolling"     | "charming"    | "smashing"      | "throwing"  | "puking"
              "talking"     | "bleeding"    | "twirling"      | "crying"    | "weeping"
              "sleeping"    | "dreaming"    | "flailing"      | "smoking"   | "shaving"
              "burning"     | "drowning"    | "farting"       | "flaming"   | "screaming"

/ Adjectives beginning with Consonant (mostly describing characteristics)

Adj ->        "amazing"     | "neurotic"    | "swell"         | "hydrostatic"
              "gooey"       | "petulant"    | "hypnotic"      | "low-calorie"
              "turquoise"   | "insipid"     | "godless"       | "ravenous"
              "psychic"     | "deadly"      | "pink"          | "luscious"
              "loud"        | "geriatric"   | "big-boned"     | "angry"
              "soft"        | "gentle"      | "diseased"      | "foamy-necked"
              "lazy"        | "rabid"       | "naked"         | "dreadful"
              "tiny"        | "supersonic"  | "pagan"         | "high-velocity"
              "blind"       | "opaque"      | "large"         | "incredible"
              "puny"        | "radical"     | "dead"          | "felonious"
              "dry"         | "lonely"      | "noxious"       | "bald"
              "motley"      | "drab"        | "dang"          | "clumsy"
```

APPENDIX C. SAMPLE OUTPUT

Three examples of bonzotext produced by the implemented system described in Chapter 3 are given in this Appendix. They were each generated using different language files.

C.1 SAMPLE BONZOTEXT 1: COMPLAINT LETTER

Dear Sir:

I am writing to express my concerns about Ted Wilcox, and more specifically, his criticisms regarding scary weirdos. It is worth noting at the outset that essentially, he is nothing if not annoying. Nonetheless, Ted should just quit whining about everything.

To start, his views are nothing short of nerdy. I imagine that nobody likes neurotic deadheads. You see, Ted is intentionally being diabolic. Let me cut to the chase: Anger is contagious. Let me mention again that the hate just keeps on coming. More prosaically, bloodthirsty imbeciles have no business here.

It's my hunch that like hate-filled dipsomaniacs, he will certainly coordinate a revolution. To be honest, his use of counter-productive neanderthals is unquestionably pathetic. On a similar note, my observations are perhaps unique. Curiously, I'm indubitably afraid of savage vigilantes. As will become apparent within a short period of time, he is wrong. Please let me explain that his claims are pure tripe.

To begin with, his theories are a pitiful jumble of incoherent nonsense. Generally speaking, I predict catastrophe. Although he is wrong, his tactics will regulate jingoism sometime soon.

Certainly, it's shabby fiends like Ted that manipulate everything and everybody. Still, anger is contagious. Moreover, subhuman authoritarians have no business here. In other words, Ted masterminded last year's now-infamous attempt to turn vegetarians loose against us good citizens.

I thank you for your attention and efforts in rectifying this situation.

Sincerely,

Adam D. Cain

This sample bonzotext was generated using a language based on S. Pakin's "Automatic Complaint Generator" [22]. The hidden message in the text above is "I think I need a burrito." It is encrypted and was bonzified with $\beta = 2$, $\rho = 3$ and $\sigma = 2$.

C.2 SAMPLE BONZOTEXT 2: MODERN POETRY

```
Dearest Emily,
    My feelings for you are impossible to express within the confines of
conventional letter-writing form.  And so I must turn to poetry.....

        "Raging and weeping women seem meaningless"

hold me... sudden movement is risky
give up thinking surely this too shall pass
architects of doom only look dangerous without complaining
you belong here -- why?  We are lost!

it appears impossible because simon says don't move
some siblings who prefer the darkness smell good where the peacocks strut
we share a secret smile: I feel partially hydrogenated!

Repack your parachute and all the teachers block our path
They often appear in your dreams
everyone notices (really it does)

you remember you cannot hide
dealers without purpose deny everything while our wounds heal
is there any truth?

Beauty and truth can't be trusted
write poems, despite the pain by starlight
Am I SHOPLIFTING?
Whenever nobody tells me anything, i like everything
without speaking... viruses among the young tumble like the autumn leaves
speak to me -- don't look over your shoulder
clouds block the moon because i am you

by lamplight waiters surround you
stop crying, but stop blaming me for everything
you're safe now

keep your shadow clean
just keep on dancing (instead of leaving)
poisoned chocolates never sleep
nobody cares what happens
next time you should have stayed in bad


Well, I hope you understand now.

       Yours sincerely,
              Adam
```

he hidden message in the bonzotext above is "Send help!" which was symmetrically encrypted sing PGP (with passphrase "password1") before bonzification with parameters $\beta = 2$, $p = 0$. he ciphertext is 46 bytes in length and was not processed with Stealth.

C.3 SAMPLE BONZOTEXT 3: RECIPES

```
Dear Adam,
   Here's a sampling of the recipes I've been developing.  If any of
them sound good let me know and I'll make it for dinner tonight.

-------------------------------------------------
Broiled linguine squares pilaf

1 guava, rinsed and drained
4 2/3 gram pork
1 pheasant
1 jalapeno, dried
5 gram linguine
1 guava
2/3 tablespoon basil
6 3/4 jar mayonnaise

Thaw ingredients.  In a large saucepan, press frozen spinach, yogurt, catsup
and chives until done.  Add peppers.  Form into patties.  Grind some mustard
seed, lima beans, broccoli and tofu until done.  Fold in broth, stirring
occasionally.  Keep covered.  Drain until tender.  Yield: 10 Servings
-------------------------------------------------
Hong Kong raisin cookies du jour
  (home-made apples give this a bit of bitterness)

1 raisin, well-scrubbed
1 2/3 quart catsup
1 1/4 gram beef
1 eggplant, dried
2/3 tablespoon coriander
1 ounce broccoli
1 raisin, thinly sliced
1/8 tablespoon pepper
1 mango
3 1/8 cup mayonnaise
1 avocado, peeled
1 1/4 quart water
1 tomato, chopped
1 apple, minced or pressed

Grease one large baking pan.  Whisk together all ingredients.  Chill until
it attains the desired consistency.  Season with mustard seed to taste.
Serve on a bed of lettuce.  Yields 5 servings.

Hope you like them.
   Joe
```

This sample bonzotext was generated using a language written by Joseph Futrelle The hidden message in the text above is "Pass the milk." It is unencrypted and was bonzified with $\beta = 2$, $\rho =$, and $\sigma = 5$.

REFERENCES

[1] W. Allen, "Sleeper," New York, NY, 1973.

[2] P. Zimmerman, *The Official PGP User's Guide*. Cambridge, MA: MIT Press, 1995.

[3] D. Kahn, *The Codebreakers*. New York, NY: Macmillan, 1967.

[4] N. F. Johnson, "Steganography," `http://www.patriot.net/users/johnson/html/neil/stegdoc/stegdoc.html`, March 1996.

[5] G. J. Simmons, "The Prisoner's Problem and the Subliminal Channel," in *Advances in Cryptology: Proceedings of Crypto '83*, D. Chaum, Ed. New York, NY: Plenum, 1984, pp. 51-67.

[6] J. L. Massey, "An Introduction to Contemporary Cryptology," in *Proceedings of the IEEE*, vol. 76, no. 5, pp. 533-549, May 1988.

[7] J. Brassil, S. Low, N. Maxemchuk, L. O'Gorman, "Electronic Marking and Identification Techniques to Discourage Document Copying," in *Proc. INFOCOM94*, vol. 3, June 1994, pp. 1278-1287. `ftp://ftp.research.att.com/dist/brassil/1994/infocom94a.ps.Z`

[8] L. Carroll, *The Humorous Verse of Lewis Carol*. New York, NY: Dover Publications, Inc., 1960.

[9] K. Maher (private communication), 1995.

[10] P. Wayner, "Mimic Functions," in *Cryptologia*, vol. 16, no. 3, pp. 193-214, July 1992.

[11] C. E. Shannon, "Prediction and entropy of printed English," in *Bell System Technical Journal*, vol. 30, pp. 50-64, Jan.1951.

[12] H. Kenner and J. O'Rourke, "A travesty generator for micros," in *Byte*, p. 129, Nov. 1984.

[13] M. V. Shaney (private communications), 1985-1996.

[14] P. Wayner, "Strong Theoretical Stegnography," in *Cryptologia*, vol. 19, no. 3, pp. 285-299, July 1995.

[15] C. E. Shannon, "Communication theory of secrecy systems," in *Bell System Technical Journal*, vol. 28, pp. 656-715, Oct.1949.

[16] R. Hoselton (private communication), April 1996.

[17] T.C. Bell, J. G. Cleary, I. H. Witten, *Text Compression*. Upper Saddle River, NJ: Prentice Hall, 1990.

[18] B. Schneier, *Applied Cryptography*, 2nd ed. New York, NY: Wiley, 1996.

[19] H. Hastur, *Stealth for PGP v1.1*, `http://www.dcs.exeter.ac.uk/~aba/stealth/`.

[20] *Gale's Quotations*. Gale Research Inc., Detroit, MI, 1995.

[21] S. Pakin, *Scott Pakin's automatic complaint-letter generator*, `http://www-csag.cs.uiuc.edu/individual/pakin/complaint`.

[22] S. Pakin (private communication), March 1996.

[23] D. E. Knuth, *Seminumerical Algorithms* in *The Art of Computer Programming*, vol. 2. Reading, MA: Addison-Wesley, 1981.

[24] R. Ganesan and A. T. Sherman, "Statistical techniques for language recognition: An introduction and guide for cryptanalysts," in *Cryptologia*, vol. 17, no. 4, pp. 321-366, Oct. 1993.

[25] R. Ganesan and A. T. Sherman, "Statistical techniques for language recognition: An empirical study using real and simulated English," in *Cryptologia*, to be published.